Physical Design for Non-relational Data Systems

by

Michael J. Mior

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: **Paolo Atzeni**
Professor, Dipartimento di Ingegneria
Università Roma Tre

Supervisor: **Kenneth Salem**
Professor, David R. Cheriton School of Computer Science
University of Waterloo

Internal Members: **Grant Weddell**
Professor, David R. Cheriton School of Computer Science
University of Waterloo

Bernard Wong
Associate Professor, David R. Cheriton School of Computer Science
University of Waterloo

Internal-External Member: **Lin Tan**
Associate Professor, Dept. of Electrical and Computer Engineering
University of Waterloo

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation. These include:

1. Kenneth Salem
2. Ashraf Aboulnaga
3. Rui Liu

Portions of this thesis are based on published or accepted peer-reviewed papers of which I am the first author [95, 96, 94]. I was the primary contributor to this work, including the conception and development of the ideas, the analysis, and the preparation of papers. The collaborators listed above provided feedback and assisted with editing the text of these papers.

Abstract

Decades of research have gone into the optimization of physical designs, query execution, and related tools for relational databases. These techniques and tools make it possible for non-expert users to make effective use of relational database management systems. However, the drive for flexible data models and increased scalability has spawned a new generation of data management systems which largely eschew the relational model. These include systems such as NoSQL databases and distributed analytics frameworks such as Apache Spark which make use of a diverse set of data models. Optimization techniques and tools developed for relational data do not directly apply in this setting. This leaves developers making use of these systems with the need to become intimately familiar with system details to obtain good performance.

We present techniques and tools for physical design for non-relational data systems. We explore two settings: NoSQL database systems and distributed analytics frameworks. While NoSQL databases often avoid explicit schema definitions, many choices on how to structure data remain. These choices can have a significant impact on application performance. The data structuring process normally requires expert knowledge of the underlying database. We present the NoSQL Schema Evaluator (NoSE). Given a target workload, NoSE provides an optimized physical design for NoSQL database applications which compares favourably to schemas designed by expert users. To enable existing applications to benefit from conceptual modeling, we also present an algorithm to recover a logical model from a denormalized database instance.

Our second setting is distributed analytics frameworks such as Apache Spark. As is the case for NoSQL databases, expert knowledge of Spark is often required to construct efficient data pipelines. In NoSQL systems, a key challenge is how to structure stored data, while in Spark, a key challenge is how to cache intermediate results. We examine a particularly common scenario in Spark which involves performing iterative analysis on an input dataset. We show that jobs written in an intuitive manner using existing Spark APIs can have poor performance. We propose ReSpark, which automates caching decisions for iterative Spark analyses. Like NoSE, ReSpark makes it possible for non-expert users to obtain good performance from a non-relational data system.

Acknowledgements

I am thankful for the continued guidance provided by my advisor Dr. Ken Salem. Without his patience and support through many years (and many drafts) I surely would not have made it this far.

Thanks also to the rest of my committee, Dr. Paolo Atzeni, Dr. Lin Tan, Dr. Grant Weddell, and Dr. Bernard Wong for their time invested in reviewing and providing feedback on this thesis.

I am also grateful for my church family at Waterloo University Bible Fellowship. Their support in countless ways over the years and their friendship has meant a great deal. They and the other friends I have made during my time in Waterloo have enriched my life significantly.

Finally, thank you to my family for their continued love and support through my many years of study. Their encouragement has been invaluable to me.

Dedication

This thesis is dedicated to my Lord and saviour Jesus Christ to whom I owe my life.

> Whatever you do, in word or in deed, do all in the name of the Lord Jesus, giving thanks to God the Father, through him. — Colossians 3:17

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Physical design is an important consideration when building data systems. The appropriate choice of data structures and caching policies can have a significant impact on performance. The problem of physical design has been studied extensively for relational databases. However, there has been little work on physical design for newer non-relational systems, despite physical design being an equally important concern. Unlike relational data systems, non-relational data systems typically lack any design tools and automation and instead rely on manual design and configuration, which demand expert knowledge of the target system. In this thesis, we explore the problem of physical design in two types of non-relational data systems: NoSQL databases and distributed data processing frameworks. In both settings, we automate decisions which previously required expert input and achieve either comparable or improved performance.

1.1 NoSQL Database Schemas

NoSQL systems have become a popular choice as database backends for applications because of the high performance, scalability, and availability that they provide. In Chapters 3 and 4 of this thesis, we focus on systems that Cattell termed *extensible record stores* in his taxonomy of NoSQL systems [29]. In extensible record stores, applications can create tables of records, with each record identified by a key. However, the application need not define the set of columns in the records in advance. Instead, each record can have an arbitrary collection of columns, each with an associated value. Because of this flexibility, applications can encode their data in both the keys and column values. We refer to tables

in such systems as *column families*. Examples of extensible record stores that support this column family model include Cassandra [79], HBase [1], and Bigtable [31].

Before a developer can build an extensible record store application, it is necessary to define a schema for the underlying record store. Although the schema of an extensible record store is flexible in the sense that the application does not need to define specific columns in advance, it is still necessary to decide what column families will exist in the record store, what information each column family will encode, and how it will be encoded. That is, whether data will be stored as keys, column names, or column values. These choices are important because the performance of the application depends strongly on the underlying schema. For example, some schemas may allow the system to provide answers to some queries with a single lookup while other queries may require multiple lookups.

Although it is important to choose a good schema, there are no tools or established methodologies to guide and support this process. Instead, schema design for extensible record stores is commonly based on general heuristics and rules of thumb. For example, eBay [108] and Netflix [76] have shared examples and guidelines for designing schemas for Cassandra. Specific recommendations include *not* designing column families as one would design relational tables, ensuring that column families reflect the anticipated workload, and denormalizing data to improve read performance. While such recommendations are useful, they are necessarily vague and generic, and require adaptation to each application.

The first research contribution of this thesis is a tool that can recommend a specific schema, optimized for a target application, so that it is not necessary to rely on general rules of thumb. We propose a principled approach to the problem of schema design for extensible record stores. Our tool, the NoSQL Schema Evaluator (NoSE) uses a cost-based approach. By estimating the performance that candidate schemas would have for the target application, we recommend the schema that results in the best estimated performance. We designed our tool for use early in the application development process; the tool recommends a schema and the application is then developed using that schema. In addition to providing a schema definition, our tool also recommends a specific implementation of the application's queries against the proposed schema. NoSE is described in Chapter 3. We show that the designs produced by NoSE can outperform those produced by expert designers. Furthermore, the use of NoSE does not require the user to understand the performance characteristics of the target database.

While NoSE is helpful for designing new applications, existing NoSQL applications using denormalized data can also benefit from automated approaches to schema design. NoSE, like many relational design tools, distinguishes between an abstract *conceptual model* of the database and a *physical design*, which describes how the database should be stored

to provide good performance for a specific application workload and target NoSQL system. Existing applications typically did not have the benefit of a tool such as NoSE and may not have a well-defined conceptual database model. Without such a model, it can be difficult for application developers to understand the database and to revise the physical design as the application evolves. For example, to add a new type of information to the database, an application developer must determine how to incorporate that information into the existing denormalized schema, depending on how the new information will be used. To add new queries or updates, the application developer must understand whether they can be supported efficiently with the existing physical database design, or determine how to change the design to support the new workload without hurting the performance of existing parts of the application.

Since denormalization is done by the application developer, the database system is unaware of it and cannot help. Unless the application developer maintains external documentation, the only knowledge of this denormalization is embedded within the source code. We aim to surface this knowledge by generating a useful conceptual model of the data from either an instance of the denormalized database or user-provided information on the database structure. We refer to this surfacing task as *schema renormalization*. Our solution to the renormalization problem is presented in Chapter 4. We present an algorithm we call ESON which is provided a denormalized database schema and functional and inclusion dependencies on that schema as input. We show both by construction and by example that ESON is able to remove denormalization which exists in the schema to produce a useful conceptual model.

1.2 Distributed Data Processing

Unlike NoSQL database systems, many distributed computing frameworks such as Apache Spark do not rely heavily on workload-optimized physical design. This is because data is commonly processed from a variety of external sources and the format of the data is not under the control of the application developer. Nevertheless, for good performance, the physical representation of intermediate results must be carefully chosen. In particular, Spark relies on effective materialization and caching of these intermediate results.

Applications using Spark's computation model have been constructed for a wide variety of domains including machine learning and graph analytics. A common pattern in applications is to iteratively evolve a dataset until reaching some user-specified convergence condition. Unfortunately, the Spark runtime is unaware of the existence of iteration in the

application program. Furthermore, Spark's execution model makes it difficult for developers unfamiliar with implementation-level details of Spark to write efficient programs.

In Spark applications, effective use of caching is necessary to avoid recomputing intermediate results from previous iterations. However, we demonstrate that straightforward use of Spark's caching APIs can unintuitively result in poor performance. In Chapter 5, we present ReSpark, a series of enhancements to the APIs provided by Spark to explicitly expose information on iterative computation to the Spark runtime. This allows ReSpark to automate caching decisions for iterative programs in a similar manner to how we automate schema design for NoSQL systems. As is the case with NoSQL applications, these decisions are often non-intuitive and would normally rely on expert knowledge of the Spark runtime. Through an evaluation against real-world applications, we demonstrate that ReSpark's caching policy can achieve comparable performance to manually optimized workflows without the need for deep understanding of Spark's caching and execution mechanisms.

1.3 Thesis Organization and Research Contributions

Chapter 2 presents related work. Our contributions are presented in the following chapters:

- Chapter 3 describes the NoSQL Schema Evaluator (NoSE), our tool for schema design automation for NoSQL databases. NoSE provides a fully automated tool for NoSQL database design. We implemented NoSE as a prototype targeting the Apache Cassandra wide column store. Our experimental results show that designs created by NoSE outperform designs created by an expert user of Cassandra.

- Our algorithm, ESON, for recovering useful normalized models from denormalized NoSQL schemas is given in Chapter 4. We demonstrate with examples that ESON is able to recover useful models from a variety of denormalized database models, in an automated or semi-automated fashion.

- In Chapter 5, we continue our exploration of automating design decisions in modern data processing systems. ReSpark automates caching for iterative workflows in Apache Spark similar to how NoSE is able to optimize NoSQL systems. By exposing iteration to the Spark runtime, ReSpark automatically decides when to cache intermediate results. A performance evaluation on several workloads shows that ReSpark can achieve performance similar to applications with manual caching annotations.

Finally Chapter 6, summarizes the thesis and provides some directions for future work.

Chapter 2

Background and Related Work

This chapter provides background on the physical design problems we consider in this thesis, as well as the non-relational data systems we aim to optimize. Section 2.1 provides an overview of different types of NoSQL databases. Solutions to physical design problems in relational databases are outlined in Section 2.2. Existing solutions for schema design for NoSQL systems as well as their shortcomings are described in Section 2.3. Sections 2.4 and 2.5 discuss work related to high-level models of data and queries relevant to our physical design problem. In Section 2.6, we summarize existing approaches for normalization of database schemas as they relate to our algorithm for schema renormalization. Finally, Section 2.7 discusses existing solutions for distributed data processing in non-relational data systems.

2.1 NoSQL Client Interfaces

The interface provided to clients by NoSQL databases is often much more restricted than a relational database interface. For any fixed schema, the client interface determines the class of queries which can be answered. Previous authors have discussed differences among the interfaces presented by NoSQL databases [22, 71]. Below we summarize some points of discussion and examine how this impacts queries which can be answered for a given schema. It is difficult to cover the wide variety of interfaces available in NoSQL systems. Our focus in the subsections below is according to the classification defined by Cattell [29].

2.1.1 Key-Value Stores

The primary interface to many key value stores consists of two operations: `put` and `get` [29]. `put` stores a value under a particular key and `get` retrieves a value given a key. This implies that if an application wants a query to be directly answerable by a key-value store, the answer to the query must be stored in the database and the application must have the corresponding key. This severely limits the types of queries which can be performed without significant denormalization by the application when keys are stored. It is common for key-value stores to support atomic operations only on individual keys. This is a result of the coordination overhead which is required for distributed transactions across multiple keys.

Many key-value stores have expanded their interface to support other useful operations. For example, Redis [9] is referred to as a "data structure server." In addition to `put` and `get`, it supports storing other data structures such as sets, hashes, and lists. HyperDex [51] also offers similar interfaces for these structures.

2.1.2 Extensible Record Stores

The interface presented by extensible record stores is typically a superset of the `put`/`get` interface of a key-value store. Apache Cassandra [79], Apache HBase [1], and Bigtable [31] all present a similar interface. Groups of records are divided into "column families" which are analogous to relational database tables. Each column family consists of a series of rows with each row identified by some arbitrary key. Rows can contain an arbitrary number of columns and data is often sparse (i.e. it is common for rows to have no values for most columns).

Clients can retrieve data from extensible record stores by specifying some range of row keys. However, in the case of the most common configuration of Cassandra, range queries across row keys are disallowed and a single row key must be given. In either case, the choice of row key is important for data locality. Data are placed on nodes in the distributed system according to their row key. A range can also be given for columns, which allows only a subset of the data corresponding to a row key to be returned.

In addition to specifying the locality and possible query patterns, the selection of row key also impacts data modification. When updating data in an extensible record store, many systems restrict the atomicity of updates to data under a single row key. This is important when designing a schema for an application which requires atomic updates. If

a data item is explicitly denormalized under multiple row keys, it will not be possible to perform atomic updates on that piece of data.

2.1.3 Document Stores

Records in a document store are referred to as "documents". Documents allow arbitrary nesting of values identified by some primary key. Documents are grouped into "collections" which typically correspond to the type of entity being modeled. This basic model is shared by many document stores such as MongoDB [8], CouchDB [5], and Amazon's SimpleDB [3]. All of these document stores support retrieving a document from a collection by its key as well as the definition of secondary indexes on properties of the document.

One feature typically lacking in document stores is joining documents between two collections. Some document stores such as SimpleDB don't provide any mechanism to perform joins. One alternative is the map-reduce [47] interface presented by databases such as MongoDB and CouchDB. This interface allows users to specify arbitrary map-reduce programs which execute inside the database engine. By writing a map-reduce program it is possible to perform joins and complex aggregations. However, map-reduce is not designed to be used in real time and can produce unacceptably high latencies for some queries [39].

As with extensible record stores, a common restriction when updating documents in a document store is that an update can only be performed atomically within a single document. If it is necessary to update data atomically, then this data must be grouped together under a single document.

2.2 Relational Database Physical Design

One formulation of the problem of physical database design is the index selection problem (ISP) [40]. The goal of the index selection problem is to find an optimal or near-optimal set of indexes for relations in a relational database given a particular workload. Exactly one primary index per relation must be selected (although this is commonly fixed as corresponding to the primary key of each relation). One or more secondary indexes which improve performance of queries in the workload may be selected. This physical design problem has also been extended to include other denormalized physical structures such as materialized views [11]. Many design tools currently exist in commercial products such as Microsoft AutoAdmin[11], DB2 Design Advisor[132], and Oracle Automatic SQL Tuning[44]. Badia and Lemire [16] break down relational database modeling into three steps: 1) conceptual

modeling (which we further discuss in Section 2.5), 2) logical modeling, and 3) physical modeling. They identify logical modeling as a well-defined process in relational schema design, although this is not the case for NoSQL databases. The ISP for relational databases (with the inclusion of materialized view selection) equates to step three above.

Vertica[80] consists of a SQL interface built on the C-Store column-oriented database[117]. It uses multiple encoding techniques to efficiently store denormalized views (called projections) to improve query performance. DBDesigner [124] is a tool for Vertica which is able to automatically select projections. To make these selections, it makes use of Vertica's query optimizer and user-defined policies to iteratively propose projections which provide some benefit to queries in the workload. While the selected schemas perform well, the iterative approach may yield suboptimal results. Earlier work by Rasin and Zodnik[111] also discusses automated schema design for C-Store. However, these techniques cannot be easily generalized to apply to NoSQL database architectures. The rest of this section analyzes common techniques to solve the ISP in a relational database setting while identifying shortcomings in translating these approaches to NoSQL databases.

2.2.1 Structure Enumeration

When performing physical schema design, it is necessary to determine which structures to evaluate. Candidate enumeration is the process of constructing a candidate set of possible physical structures which may be useful in a final schema. Enumeration is normally performed using simple heuristics such as constructing materialized views for queries in the input workload [133]. The major challenge for the DB2 Design Advisor was the large search space produced by interactions between various features (e.g. materialized views and indexes). Design Advisor defines an algorithm called SAEFIS (Smart column Enumeration for Index Scans) [115]. SAEFIS avoids enumerating all possible indexes for a query by examining various combinations of attributes used in equality and range predicates, ordering clauses, and other columns in the query. This significantly reduces the number of secondary indexes which are examined. For constructing materialized views, more advanced techniques can examine common subexpressions within queries in the input workload which are used to construct views [41].

In the NoSQL setting, enumeration is still required in order to provide options on which structures should be consider. However, due to the lack of a standard query language and differences in physical data structures, existing techniques for enumeration in relational databases do not apply directly.

2.2.2 What-If Analysis

After determining a set of candidate structures, physical schema design tools typically rely on a "what-if" analysis [33]. These analyses may make use of the database query optimizer to estimate the cost of executing queries using physical structures such as materialized views or secondary indexes without the overhead of actually creating these structures. The use of what-if analysis allows the design tool to obtain an estimated cost for using each candidate physical structure. We note however, that this is generally not possible in schema design for NoSQL databases as there is no optimizer for higher-level application queries. With a cost estimate, a simple solution used by some advisors is to construct a variant of the knapsack problem and to maximize the expected reduction in cost for a given storage budget [115].

Calls to the optimizer for query cost estimates can be the limiting factor in algorithm performance due to the high latency of these calls. C-PQO [20] reduces the number of calls to the optimizer to one per query in the input workload. This approach works by instrumenting the optimizer to isolate rules which are relevant to access path selection. This allows a single call to the optimizer to produce the possible access paths to use for a query. When solving the optimization problem, it is then possible to simulate single-index query execution plans based on the available access paths. Using these simulated plans, the original index selection problem can be solved. INUM [104] also attempts to reduce the number of calls to the optimizer by caching the results of optimizer calls. The INUM approach can be used with any index selection algorithm to improve execution speed. After calling the optimizer once for a query, INUM simply replaces the data access operators in the query execution plans with operators using a different physical structure. It is possible to estimate the cost of these operators relying only on the cost model used by the optimizer without requiring the entire optimization procedure to be invoked.

Several alternative solutions have been developed with the goal of either producing higher-quality results or producing results with a faster execution time. Bruno and Chaudhuri [19] take a relaxation-based approach. The initial set of candidates is the union of the optimal configurations for each query in the input workload. They then construct a search space by defining several merge operators which can combine or remove physical structures from a set of candidates. The goal is to produce a new candidate physical schema which requires less space without a significant loss of efficiency (in terms of expected query execution time). Candidate schemas are then recursively "relaxed" by replacing physical structures to produce new candidates until a time bound is exceeded. The recommended physical design is then the one with the least expected cost among those which have been discovered.

However, none of these solutions are directly applicable in the NoSQL database setting since NoSQL databases typically have no cost model. This makes it impossible to estimate query execution times. While we can develop a useful cost model for NoSQL databases, any selection algorithm in a NoSQL setting will be required to select structures based on a conceptual model. NoSQL database design requires selection of the entirety of the structures to be stored in the database and cannot assume the existence of any structures such as base tables which is common for relational database design tools. This differs from the common case for relational databases described above where the goal is to select secondary structures over an existing model.

2.2.3 Design Optimality

Many efforts in constructing design tools have also attempted to find globally optimal solutions. One approach is to use genetic algorithms to mutate possible design decisions until an acceptable solution is found. A desirable property of this approach is that continued execution will lead to better solutions. Kołaczkowski and Rybiński propose an approach which starts with the optimal execution plan for each query and then mutates operators in the plan space [77]. Calle et al. take a similar approach but consider mutating a selection of physical structures [25]. Both of these approaches show significant improvements when compared to commercial database optimizer tools.

Other work has focused on formulating improved linear programming solutions to the problem. CoPhy [45] uses an integer linear program (ILP) with constraints that each query in the original workload must have exactly one valid plan. The objective function used is the expected cost of executing queries in the workload under a particular selection of physical structures. As with the genetic algorithm approach, the solver may be terminated as soon as an acceptable solution is reached but may be run continually until optimality if desired. Formulation as an ILP is also desirable since it enables the addition of additional constraints desired by the database administrator. An example is that the update cost for some physical structures should not exceed a particular threshold. We make use of a similar ILP formulation in our automated NoSQL schema design tool described in Chapter 3.

Relational tools, as described above, rely on input from the database system's query optimizer. In many NoSQL systems, the simple data model and query language means that an optimizer for higher level queries does not exist. This increases the importance of effective schema design since there are fewer opportunities for optimization given a fixed schema and a limited number of access paths.

2.3 NoSQL Schema Design

Although the problem of schema design for NoSQL databases is relatively new, there has been some existing work exploring this problem. The remainder of this section discusses existing solutions and highlights opportunities for improvement.

2.3.1 Design Principles

Users of NoSQL databases frequently rely on manually applied "rules of thumb" for schema design. These approaches are generic and rely on the expertise of the application designer to select a suitable schema. To illustrate this point further, one commonly cited piece of advice for the Cassandra extensible store is to model physical structures to closely correspond to the application workload [108, 76]. This frequently involves some level of denormalization. For example, consider an e-commerce application which allows users to "like" items [108]. If an application frequently retrieves a list of users who like an item when the item is displayed, then it is beneficial to denormalize and store user data partitioned by items. On the other hand, if it is more common to retrieve a list of items a user has liked, then item data should be denormalized and partitioned by user. While these rules of thumb are helpful, they are often vague and self-contradictory. One recommendation given by Patel [108] to denormalize and duplicate for read performance immediately followed with a warning not to do so unless necessary. In fact, correct application of these rules can rely on expert knowledge of performance of the backend database to know what denormalizations will be beneficial in different scenarios. In the example above, the user must know that it is inefficient to separately request information on users who like an item if that data is not denormalized.

A similar problem exists with document store databases. For document store schema design, a common design decision is embedding versus external references [2]. Documents can be nested arbitrarily deep in a document store, so it is possible to embed one entity within the document corresponding to another entity. Alternatively, a document can contain a reference to the primary key of another document. Since joins cannot be performed efficiently, the choice of embedding has a significant impact on performance. Kanade et al. [69] studied the effects of this design choice for a simple data model and workload. They observed a difference of several orders of magnitude in query execution time depending on the level of embedding. Moreover, the performance differences vary dramatically for different queries suggesting that knowledge of the workload is critical in making the choice of embedding.

2.3.2 Automated Approaches

NoSQL databases present several new issues which are not solved by relational schema design tools. There is not a clear separation of the application data model and physical schema in a NoSQL database. In many cases, the database has no awareness of the data model actually used by applications since the database only sees a denormalized view of the data. This leaves any mapping between the application data model and the physical schema dependent on knowledge of the database administrator. This is evidenced by various developers attempting to evaluate application changes when moving from relational to NoSQL database backends. Project EPIC [114] found that significant changes from a relational data model were required when moving to the Cassandra extensible record store. The authors discovered that this required significant denormalization to efficiently service different application queries. Their analysis was heavily dependent on knowledge of the queries executed by the application and required the application to change in order to make use of the denormalized data.

Scherzinger et al. [113] also identify the need for good schema design in NoSQL stores by evaluating the high cost of poor schema design. Data items are frequently colocated in order to reduce the number of distributed transactions. However, the more data is grouped together, the higher the likelihood of concurrent writes to the same partition of data. A naïve schema design can result in 20–35% of write transactions failing for a given workload. This is a result of the high level of concurrent writes and the use of optimistic currency control by the backend database. The problem is completely avoided by selecting a better schema. Kanade et al. [69] examine a related problem in the MongoDB document store. They identify multiple techniques for representing a data model in a document store and show that the representation can have an order of magnitude impact on query performance. Furthermore, the impact is highly dependent on the particular workload. They use this observation as motivation for exploring workload-driven physical design for document stores.

An existing tool, by Vajk et al.[123], also attempts to solve the problem of NoSQL database design. Their system starts with a normalized schema and produces an optimized schema based on the cost of executing a given set of queries. The authors identify a tradeoff between query execution time and the storage cost of denormalization which is relevant to development of our automated schema design tool which we introduce in Chapter 3. The approach does not deal with updates, which we do address in our work.

Rule-based approaches have also been proposed for adapting relational [83] and OLAP [37, 36] schemas for NoSQL databases. Such transformations may be helpful when adapting an existing application to use a NoSQL database. However, when designing a new application

from scratch, such an approach is unnecessarily restrictive since starting from an existing schema for another database technology limits the possible schemas which can be explored. A rule-based approach also makes it difficult to incorporate workload information which is critical for schema design since data should be stored in a similar manner to how it will be accessed [108, 76].

Some rule-based approaches focus on maintaining atomicity for transactions within the original workload. As discussed further in Section 2.1, the choice of physical schema implicitly establishes boundaries for transactions. Data denormalized across these transaction boundaries cannot be updated atomically. Bugiotti et al. [23] define NoAM, the NoSQL Abstract Model. The schema design problem associated with NoAM is the partitioning of "aggregates" or collections of related entries. Aggregate partitioning attempts to colocate entries which are frequently accessed together while separating entries which are modified independently to avoid contention. This aggregate approach does not consider denormalizing or duplicating data which may be necessary when different queries access data based on distinct attributes.

Other approaches have attempted to target specific databases. SimpleSQL[24] provides a relational layer on top of Amazon SimpleDB which automatically creates the necessary schema to answer queries. However, SimpleSQL makes specific assumptions about the data model and consistency properties of the underlying datastore. This makes the approach difficult to generalize for other datastores.

SAGE [86] performs logical and physical design for "new SQL" systems. New SQL systems aim to provide the scalability of NoSQL databases while retaining the consistency and high-level query semantics of relational databases. SAGE attempts to create "entity groups" by partitioning logical entities into hierarchies of related objects so related entities can be accessed together (e.g. a customer along with all their orders). This approach works well for the class of systems SAGE was designed for, but it is most similar to the relational database techniques discussed in Section 2.2. However, the approach taken by SAGE does not translate well to NoSQL schema design as it fails to deal with important issues such as denormalization.

2.4 Query Processing in NoSQL Systems

In the following sections we examine different approaches to query processing in NoSQL systems. As discussed in Section 2.1, NoSQL databases often present highly simplified query interfaces to applications. This often requires applications using complex queries

to implement these queries manually against these interfaces. The following sections discuss how these different approaches relate to our goal of building a NoSQL design tool which supports application developers in designing schemas suitable for executing high-level queries.

2.4.1 Object-Oriented Data Access

A common abstraction layer over relational databases is object-relational mapping (ORM) [72]. ORM interfaces aim to take classes in an object-oriented programming language and add methods which automatically fetch and mutate data from a backend database. This has the desirable property of freeing application developers from the details of querying the backend database. In addition, the ORM layer can abstract away differences between backend database implementations.

ORM interfaces have also been proposed for abstracting the differences between NoSQL systems. Atzeni et al. [15] created Save Our Systems (SOS) as a simple approach to abstracting access to NoSQL systems. Their approach presents a simple interface to store and retrieve objects from different types of NoSQL stores. Störl et al. [118] present many other existing NoSQL ORM tools and show that they vary widely in their features and backend database support. They also show that these ORMs approximately double execution time over a number of different queries. One important point the authors note is that the use of an ORM implicitly defines the schema used by the backend database. This is critical to schema design as the choice of how objects are modeled also defines the schema. Kundera [7], which is the most full-featured option discussed, provides minimal support for exploiting materialized views and requires an external service to perform some complex queries such as joins and aggregation. These tools cannot effectively make use of the rule-based wisdom discussed in Section 2.3.1 of storing data of related entities together in the backend database.

2.4.2 Higher Level Query Languages

NoSQL databases commonly provide a very restricted query interface in order to achieve high performance. For example, many key-value stores, such as Riak [10], provide little more than access to a blob of data given a key. In most cases, it is either not possible or very expensive to query data by any attribute other than the key. This means that when designing the schema, care must be taken when selecting the keys since this choice defines which queries can be made efficiently in the future. Data stores following the Bigtable [31]

model offer richer semantics than simply querying by key, but still provide far fewer access paths to data as compared to relational databases. Still other databases, such as Redis [9], provide many alternative data structures such as sets and lists which can be accessed and modified by a key. These features can be exploited to efficiently support application workloads, but doing so requires strong support from the application side to compose the simple structures provided by the database. This composition requires an effective use of these data structures through a carefully-constructed application schema.

Many approaches exist for executing complex analytical queries over NoSQL databases. A popular solution for databases such as HBase and Cassandra is Hive [121]. Hive takes complex queries in a SQL-like language (HiveQL) as input and uses the MapReduce [47] framework to execute these queries. Query execution in Hive consists of executing one or more map-reduce jobs which compute values for each relevant row in the input tables and then aggregating these results to provide the query response. MapReduce targets batch jobs and as Hive queries can consist of multiple MapReduce jobs, the observed latency can be quite high. QMapper [126] aims to translate SQL queries written for a relational database into efficient and correct HiveQL queries. While QMapper uses a cost-based optimization model to reduce latency, the queries being examined still have execution times on the order of hundreds of seconds. Pig Latin [101] also attempts to provide a higher-level interface for processing data using MapReduce, but does not attempt to perform any optimization and formulates "queries" as query execution plans. These approaches are useful but typically rely on non-standard query languages.

Gadkari et al.[55] and Apache Phoenix [6] aim to support low-latency queries in HBase by compiling SQL queries into a series of HBase scans. While this can be effective, the performance of queries is still dependent on how data is partitioned. These approaches also currently fail to take advantage of data which is explicitly denormalized or pre-aggregated by the application as these changes are not visible in the data model. However, executions engines such as these would be useful for implementing query plans based on recommendations from a schema design tool.

2.5 Conceptual Modeling

Before discussing how to construct queries, it is necessary to have a conceptual model of underlying data which these queries can reference. A common data model used for expressing conceptual queries is the enhanced entity-relationship diagram (EER) [49]. EER diagrams specify a class hierarchy for entities to be stored in the database. Each possible class has an associated set of attributes as well as relationships to other classes.

In addition, it is helpful to estimate statistics for the data to be stored in the database. For example, such statistics may include the number each type of entity and the cardinality of relationships and attributes. This information is commonly exploited by query optimizers to estimating the cost of a query execution plan. The same information is also relevant to physical schema design which often makes use of the query optimizer, as discussed below. A similar approach is to use unified modeling language (UML) [112] diagrams for conceptual models. UML class diagrams are similar to EER diagrams in that they specify classes in the data model as well as attributes associated with each class. UML has also been successfully used a conceptual model when designing database schemas [100].

Many other approaches to conceptual modeling for data and queries already exist. One example is GMAP [122], which was proposed as a technique for improving the physical data independence of relational database systems. In GMAP, both application queries and physical structures are described using a conceptual entity-relationship model. Queries are mapped to one or more physical structures which can be used to produce answers to the query. In the case of GMAP, this approach is used out of a desire to provide a more thorough form of physical data independence.

Others have also proposed to write application queries directly against a conceptual schema. For example, ERQL [81], is a conceptual query language over EER models. ERQL is based around the concept of "path expressions". A path expression follows links in the EER schema to connect entity sets to either their associated attributes or to other entity sets via relationships. Predicates can then be applied to these path expressions to restrict the data returned to only contain entities matching these predicates. The authors also discuss more advanced features such as quantification and aggregate functions. They also define a view definition language which is useful for representing denormalized database structures without a specific physical schema. Kifer et al. [73] also define a query language based on path expressions, but for object-oriented databases. Similarly to ERQL, XSQL queries specify path expressions over the object graph and use these expression both for attributes and fields which are selected.

A different approach to conceptual modeling is demonstrated by query languages such as LISA-D [120] and RIDL [92] which are instead based on object role modeling (ORM). ORM simply considers the conceptual model as a set of objects playing various roles. These languages were very complex, with others such as ConQuer [18] being developed with the primary goal of being easily understood by end users. A transformation process allows ConQuer queries to be translated into SQL queries for a given relational model.

Other related work exists in database reverse engineering (DBRE). The goal of DBRE is to produce a greater understanding of the semantics of a database instance, commonly

through the construction of a higher level logical model of the data. Some approaches, such of that of Andersson [12], depend heavily on information extracted from join queries to determine related entity sets. Such queries do not exist in our setting since NoSQL databases do not permit joins. Others, such as those proposed by Chiang et al. [38] and Premerlani and Blaha [109], present only an informal process as opposed to a specific algorithm. Markowitz and Makowsky [90] present a technique for converting relational database schemas into enhanced entity relationship (EER) schemas [119]. Their algorithm $\boldsymbol{Rmap}^R$ accepts a relational schema and a set of functional and inclusion dependencies and produces an EER schema. However, their approach does not produce a schema normalized according to the input functional dependencies.

Conceptual query languages, such as those defined above, present useful tools for schema design. We can borrow concepts from these languages and adapt them for our purposes. Our current approach most resembles GMAP with adaptations to make it more suitable for use in Cassandra. Our schema design tool begins with a conceptual model and queries are written against this conceptual model in a similar manner to ERQL. When performing schema extraction, we aim to produce a conceptual model in a form similar to that of the EER models used by ERQL. However, we restrict queries to those which we know can be easily and efficiently answered using structures which fit into common NoSQL database systems (we specifically focus on Apache Cassandra).

2.6 Database Normalization

Significant literature exists dealing with database normalization. Much of this existing work revolves around eliminating redundancy in relational tables based on different forms of data dependencies. Codd [42] defines normal forms of relations based on functional dependencies. Eliminating redundancies based on functional dependencies is a necessary but not sufficient condition for normalization in our setting. Codd's normal forms focus around decomposing relations to eliminate redundant information. However, they do not deal with the case where the application duplicates data across multiple relations.

Inclusion dependencies are a natural way to express duplication of data in multiple relations. Other researchers have established normal forms which aim to normalize relations according to inclusion dependencies [88, 85, 82] in addition to functional dependencies. These normal forms, specifically IDNF [82], provide the foundation of our approach to renormalization discussed in Chapter 4. Although these normal forms are useful, there has been little work in algorithms for normalization. Our approach borrows heavily from

Mannila and Räihä [88] who present a variant of a normal form involving inclusion dependencies and an associated interactive normalization algorithm. However, it does not produce useful schemas in the presence of heavily denormalized data which we established as common in our setting. Specifically, their approach is not able to eliminate all data duplicated in different relations.

Also related is the problem of *schema matching* [110] where the goal is to establish the correspondence between two schemas. A solution to the schema matching problem would allow lifting an application on a given logical schema, but is not useful for discovering which logical schema to use, a nontrivial task which we address in this work.

There is significant existing work in automatically mining both functional [89] and inclusion [70] dependencies from both database instances and queries. These approaches are complementary to our techniques since we can provide the mined dependencies as input into our algorithm. We can also take advantages of faster approaches to dependency mining as they are developed. Papenbrock and Naumann [106] present a set of heuristics for making use of mined dependencies to normalize a schema according to BCNF. We leverage these heuristics to incorporate dependency mining into our algorithm as discussed in Section 4.5.

2.7 Non-relational Data Processing

Opportunities for optimizing iterative computations in distributed computing frameworks have been explored in the past with Hadoop and MapReduce [47]. HaLoop [21] modifies the MapReduce programming model to allow developers to explicitly express iterative computation. The runtime of these iterative jobs is then optimized by colocating tasks operating on the same partition on the same physical nodes across iterations. Furthermore, the data used by each of these tasks is cached on each node. While this provided significant speedup for MapReduce jobs, we note that the scheduling and caching policies of Spark are able to provide a similar level of optimization without specific awareness of iteration. In addition to scheduling optimizations, iMapReduce [131] allows map tasks to run asynchronously within an iteration. Since computation in Spark is lazy by definition, this explicit optimization is unnecessary. iHadoop [50] further exploits the potential of asynchrony by also performing the loop termination check in parallel with speculative execution of the following iteration. If the algorithm is deemed to have terminated, the additional iteration is aborted. Since computations in Spark are lazy, we note that attempting the same optimization may prove harmful. The results of the previous iteration may not be materialized and performing the termination check in parallel may result in redundant

evaluation since the Spark scheduler does not attempt to avoid duplicate executions. We leave further exploration of this technique to future work.

Several other pieces of existing work have attempted to optimize the use of the cache in Spark. Neutrino [128] attempts to optimize the use of the cache in Spark by using a dynamic programming algorithm to select the appropriate caching strategy for each partition in an RDD. Neutrino requires a trace of the algorithm execution in order to make these decisions, which we avoid in our solution (presented in Chapter 5). The authors claim the trace can be obtained from running the same algorithm on a smaller dataset. However, in some iterative Spark applications, the structure of a job is data-dependent since loop termination conditions can depend on data values. RDDShare [32] aims to identify views which can be cached in submitted Spark SQL queries to improve performance of future queries. This does not allow for optimization within a single job and only works with SQL queries. Quartet [48] has a goal similar to RDDShare of optimizing the sharing of data across jobs and functions with any Spark job. However, Quartet only serves to optimize the use of cached partitions of files and does not accelerate jobs using in-memory data. LCS [57] is a new cache eviction strategy for Spark that attempts to make more efficient use of the cache by keeping partitions which are expected to be more expensive to compute. While this results in more effective use of RDDs which have been annotated by application programmers, it does not help solve the problem of deciding what to put in the cache. Furthermore, LCS still suffers from the `unpersist` issue discussed in Section 5.3.2.

Apache Flink [26] is designed to support batch and stream processing in a single execution engine. Flink includes support for *iteration steps*, an operator which expresses iteration explicitly to the runtime in a manner similar to the one we use in ReSpark. When using Flink's iteration steps, only data from the immediately preceding iteration can be used. This is unlike Spark where each iteration can make use of arbitrary references to previously computed data. For applications fitting this pattern, having a complete view of the application structure including iteration enables Flink to determine when a dataset used during iteration is constant. When this occurs, Flink automatically caches the data so it is not recomputed on future iterations. Furthermore, Flink holds the incremental results produced by iterative computations in memory. This allows the next iteration to read this data multiple times without recomputation. While the approach to caching iteration steps used by Flink is effective, the computation model is also more restrictive. In addition to caching, Flink attempts to optimize job scheduling. Similar scheduling optimizations to scheduling in Spark may prove beneficial.

Chapter 3

Physical Schema Design Automation

Our first avenue of exploring physical design for non-relational data systems is NoSQL database systems. This chapter presents the following contributions. First, we formulate the *schema design problem* for extensible record stores. Second, we propose a solution to the schema design problem embodied in a schema design advisor we call *NoSE*, the *NoSQL Schema Evaluator*. We start with a conceptual model of the data required by a target application as well as a description of how the application will use the data. NoSE then recommends both an extensible record store schema, i.e., a set of column family definitions optimized for the target application, and guidelines for developing applications using this schema. Finally, we present a case study and evaluation of NoSE, using two application scenarios.

A conference submission based on this work was presented at the IEEE 2016 International Conference on Data Engineering [95]. An invited extended journal submission was also published in IEEE Transactions on Knowledge and Data Engineering [96].

3.1 Schema Design Example

In this section we present a simple example to illustrate the schema design problem for extensible record stores. Suppose we are building an application to manage hotel reservations. The conceptual model in Figure 3.1, adapted from Hewitt [64], describes the application's data.

The schema design problem for extensible record stores is the problem of deciding what column families to create and what information to store in each column family, for

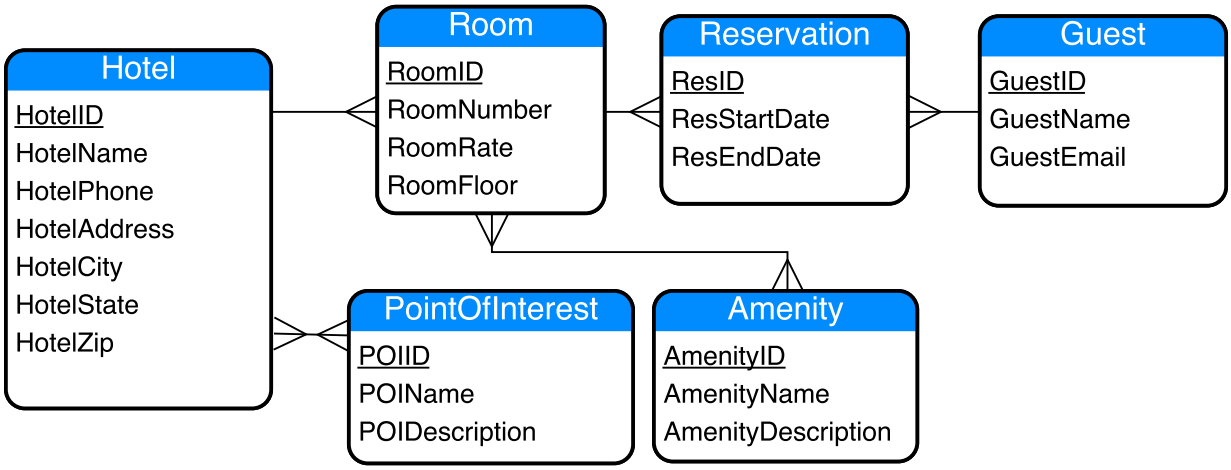


Figure 3.1: Entity graph for a hotel booking system.

Each box represents an entity set, and edges between boxes represent relationships.

a given application. In general, this will depend on what the target application needs to do. For example, suppose that the application will need to use the extensible record store to obtain information about the points of interest (POIs) near hotels booked by a guest, given the guest's `GuestID`. The primary operations supported by an extensible record store are retrieval (`get`) or update (`put`) of one or more columns from a record, given a record key. Thus, an application could easily answer this query if the record store included a column family with `GuestIDs` as record keys and columns corresponding to POIs. That is, the column family would include one record for each guest. A guest's record includes one column for each POI associated with a hotel at which that guest has booked a room. The column names are `POIIDs`, and each column stores a composite value consisting of `POIName` and `POIDescription`. In general, each guest's record in this column family may have different columns. Furthermore, the application may add or remove columns from a guest's record when updating that guest's hotel bookings in the record store. With such a column family, the application can obtain point of interest information for a given guest using a single `get` operation. This column family is effectively a materialized view which stores the result of the application query for all guests.

In this thesis, we will describe such a column family using the following triple notation:

```
[GuestID][POIID][POIName, POIDescription]
```

The first element of the triple indicates the attribute values used as record keys in the column family. The second element indicates the attribute values used as column names, and the third indicates those used as column values. We refer to the first element as the *partitioning key*, since extensible record stores typically horizontally partition column families based on the record keys. We refer to the second element as the *clustering key*, since extensible record stores typically physically cluster each record's columns by column name. We assume records in each partition are sorted according to the clustering key. Both the second and third components of the triple can optionally be empty indicating no fields in that component of the column family. The correspondence of these triples to structures used in an extensible record store is described in Section 3.2.3.

Although this column family is ideal for executing the single application query we have considered, it may not be ideal when we consider the application's entire workload. For example, if the application expects to be updating the names and descriptions of points of interest frequently, the above column family may not be ideal because of the denormalization of point of interest information, i.e., the name and description of a POI may appear multiple times in the records for different guests. Instead, it may be better to create two column families, as follows:

```
[GuestID][POIID][]
[POIID][][POIName, POIDescription]
```

This stores information about each point of interest once, in a separate column family, making it easy to update. Similarly, if the application also needs to perform another query that returns information about the points of interest near a given hotel, it may be better to create three column families, such as these:

```
[GuestID][HotelID][]
[HotelID][POIID][]
[POIID][][POIName, POIDescription]
```

In this schema, which is more normalized, records in the third column family consist of a key (a `POIID`) and a single column which stores the `POIName` and `POIDescription` as a composite value. The second column family, which maps `HotelID`s to `POIID`s, will be useful for both the original query and the new one.

The goal of our system, NoSE, is to explore this space of alternatives and recommend a good set of column families, taking into account both the entire application workload and the characteristics of the extensible record store.

NoSE solves a schema design problem similar to the problem of schema design for relational databases. However, there are also significant differences between the two problems. Relational systems provide a clean separation between logical and physical schemas. Standard procedures exist for translating a conceptual model, as in Figure 3.1, to a normalized logical relational schema, i.e., a set of table definitions, usable for defining the application's workload. The logical schema often determines a physical schema consisting of a set of base tables. The physical layout of these base tables is then optimized and supplemented with additional physical structures, such as indexes and materialized views, to tune the physical design to the anticipated workload. There are many tools for recommending a good set of physical structures for a given workload as discussed in Section 2.2.

Extensible record stores, in contrast, do not provide a clean separation between logical and physical design. There is only a single schema, which is both logical and physical. Thus, NoSE starts with the conceptual model, and produces both a recommended schema and plans for implementing the application against the schema. Further, the schema recommended by NoSE represents the entire schema, not a supplement to a fixed set of base tables. Unlike most relational physical design tools, NoSE must ensure that the workload is *covered*, i.e., that the column families it recommends are sufficient to allow for the implementation of the entire workload.

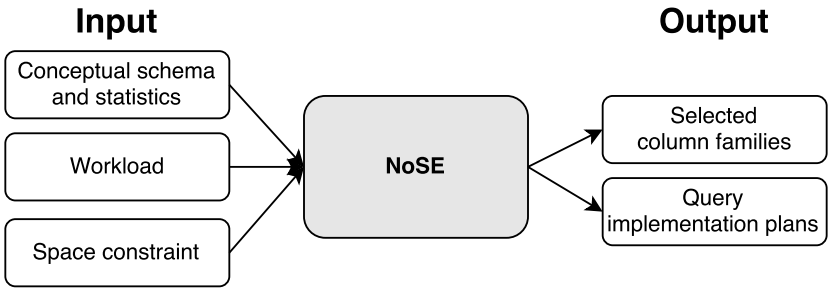


Figure 3.2: Schema advisor overview

3.2 System Overview

Figure 3.2 gives a high level illustration of the NoSE schema advisor. We designed NoSE for use early in the process of developing an extensible record store application. The advisor produces two outputs. The first is a recommended *schema*, which describes the column families used to store the application's data. The second output is a set of *plans*, one plan for each query and update in the workload. Each plan describes how the application should use the column families in the recommended schema to implement a query or an update. These plans serve as a guide for the application developer.

3.2.1 Database Conceptual Model

To recommend a schema for the target application, NoSE must have a conceptual model describing the information to store in the record store. NoSE expects this conceptual model in the form of an *entity graph*, such as the one shown in Figure 3.1. Entity graphs are simply a slightly modified (restricted) type of entity-relationship (ER) model [34]. Each box represents a type of entity, and each edge represents a relationship between entities and describes the associated cardinality of the relationship (one-to-many, one-to-one, or many-to-many). Entities have attributes, with one of these serving as a key (i.e., no composite keys are permitted). For example, the model shown in Figure 3.1 indicates that each room has a room number and rate. In addition, each room is associated with a hotel and set of reservations.

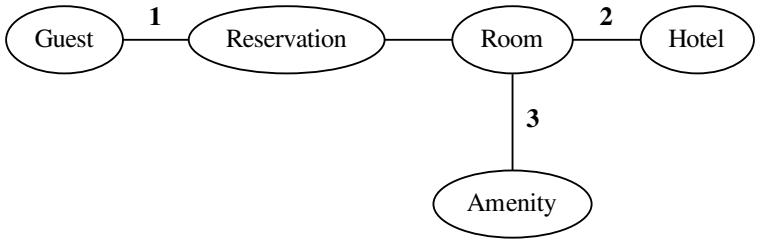
3.2.2 Workload Description

The second input to NoSE is a set of parameterized queries and updates which describe how the target application intends to use the database. Each query and update has an associated weight indicating its relative frequency in the anticipated workload. We focus here on the queries, and defer discussion of updates to Section 3.6.

NoSE expresses queries directly over the conceptual model. Each query in the workload returns information about one or more entities in the entity graph. Figure 3.3a shows an example of a NoSE query, expressed using an SQL-like syntax, which returns the names and email addresses of guests who have reserved rooms with a particular amenity in given city at a given minimum rate. In this example, `?city`, `?amenity` and `?rate` are parameters. Each query implies a *query graph* which is a subgraph of the entity graph. A path referenced in the `FROM` clause defines a portion of this graph. A query can define branches in the

```
SELECT Guest.GuestName, Guest.GuestEmail FROM Guest.Reservation.Room.Hotel
  WHERE Hotel.HotelCity = ?city AND Room.Amenity.AmenityName = ?amenity
  AND Room.RoomRate > ?rate
```

(a) Query expressed in the language used by NoSE



(b) Query graph for the query above. We describe the edge labels in Section 3.3.1.

Figure 3.3: Example query against the hotel booking system schema

graph by specifying additional paths for attributes in the `SELECT` or `WHERE` clauses. The current implementation of NoSE is restricted to acyclic query graphs. However, this is not a fundamental limitation to our approach. Queries are required to have an equality predicate on the first entity set in the `FROM` clause in order to construct a valid `get` request to the underlying datastore. Figure 3.3b shows an example of a query graph.

To define the semantics of these queries, we must consider which tuples a query produces. Conceptually, we consider the tuples produced by the join of all the relations in the query graph using the associated relationships between entity sets. These tuples are then filtered using the predicates given in the query. This means that a query returns data about a particular entity if there exists a series of related entities in the query graph that together satisfy the predicate of the query. Query results retain any duplicates in the resulting list of tuples.

We emphasize that the underlying extensible record store supports only simple `get` and `put` operations on column families, and is unable to directly interpret or execute queries like the one shown in Figure 3.3a. Instead, the application itself must implement such queries, typically using a series of `get` operations, perhaps combined with application-implemented filtering, sorting, or joining of results. Nonetheless, by describing the workload to NoSE in this way, the application developer can convey the purpose of a sequence of low-level operations, allowing NoSE to optimize over the scope of entire high-level queries, rather than being restricted to individual low-level optimizations. Of course, another problem with describing the application workload to NoSE in terms of `get` and `put` operations on

column families is that the column families are not known. Indeed, the purpose of NoSE is to recommend a suitable set of column families for the target application.

Although it is not shown in Figure 3.3a, NoSE queries can also specify a desired ordering on the query results, using an `ORDER BY` clause. This allows NoSE to recommend column families which exploit the implicit ordering of clustering keys to produce results in the desired order. A complete specification of the syntax of the query language is given below. Bracketed components are optional.

```
SELECT Attr, [Attr, ...] FROM Path
  WHERE Attr = ? [AND Attr (=|>|<|>=|<=) ?]
  [ORDER BY Attr, ...] [LIMIT n]
```

The `Path` in the `FROM` clause of the query is a path through the entity graph in the conceptual model. Each `Attr` referenced in the query references an attribute in the conceptual model. These references specify a path in the entity graph whose root is an entity specified in the `Path` in the `FROM` clause. As mentioned previously, the query graph constructed by connecting the paths of all `Attr`s to the `Path` in the `FROM` clause must be acyclic. A final additional restriction is that the `Attr` referenced in the first predicate in the `WHERE` clause must be from the entity which is the first component of the `Path` in the `FROM` clause.

3.2.3 Extensible Record Stores

The target of our system is extensible record stores, such as Cassandra or HBase. These systems store collections of keyed records in column families. Records in a collection need not all have the same columns.

Given a domain $\mathcal{K}$ of partition keys, an ordered domain $\mathcal{C}$ of clustering keys, and a domain $\mathcal{V}$ of column values, we model a column family as a table of the form

$$\mathcal{K} \mapsto (\mathcal{C} \mapsto \mathcal{V})$$

That is, a column family maps a partition key to a set of clustering keys, each of which maps to a value. Clustering keys provide an order for records in a single partition. For example, in Section 3.1, we used an example of a column family with `GuestID`s as partition keys, `POIID`s as clustering keys, and POI names and descriptions as values. Such a column family would have one record for each `GuestID`, with POI information for that guest's records clustered using the `POIID`. In short, $\mathcal{K}$, $\mathcal{C}$, and $\mathcal{V}$ correspond to the three components of the triple previously discussed.

We assume that the extensible record store supports only `put`, `get`, and `delete` operations on column families. To perform a `get` operation, the application must supply a partition key and a range of clustering key values. The `get` operation returns all $\mathcal{C} \mapsto \mathcal{V}$ pairs within the specified clustering key range, for the record identified by the partition key. For example, the application could use a `get` operation to retrieve information about the points of interest associated with a given `GuestID`. Similarly, a `put`/`delete` operation can modify/delete the $\mathcal{C} \mapsto \mathcal{V}$ pairs associated with a single partition key.

Some extensible record stores provide additional capabilities beyond the three basic operations we have described. For example, in HBase it is possible to get information for a range of partition keys, since records are also sorted based on their partition key. As another example, Cassandra provides a limited form of secondary indexing, allowing applications to select records by something other than the partitioning key. However, Cassandra applications rarely use secondary indexes for performance reasons [46]. For simplicity, we restrict ourselves to the simple `get`/`put` model we have described, as it captures common functionality.

3.2.4 The Schema Design Problem

A schema for an extensible record store consists of a set of column family definitions. Each column family has a name as an identifier and its definition includes the domains of partition keys, clustering keys, and column values used in that column family.

Given a conceptual model (optionally with statistics describing data distribution), an application workload, and an optional space constraint, the schema design problem is to recommend a schema such that (a) each query in the workload is answerable using one or more `get` requests to column families in the schema, (b) the weighted total cost of answering the queries is minimized, and optionally (c) the aggregate size of the recommended column families is within a given space constraint. Solving this optimization problem is the objective of our schema advisor. In addition to the schema, for each query in the workload, NoSE recommends a specific *plan* for obtaining an answer to that query using the recommended schema. We discuss these plans further in Section 3.3.2.

3.3 Schema Advisor

Given an application's conceptual model and workload, as shown in Figure 3.2, NoSE proceeds through four steps:

1. **Candidate Enumeration** Generate a set of *candidate* column families, based on the workload. By inspecting the workload, the advisor generates only candidates which may be useful for answering the queries in the workload.

2. **Query Planning** Generate a space of possible implementation plans for each query. These plans make use of the candidate column families produced in the first step.

3. **Schema Optimization** Generate a binary integer program (BIP) from the candidates and plan spaces. The BIP is then given to an off-the-shelf solver (we have chosen to use Gurobi [63]) which chooses a set of column families that minimizes the cost of answering the queries.

4. **Plan Recommendation** Choose a single plan from the plan space of each query to be the recommended implementation plan for that query based on the column families selected by the optimizer.

Figure 3.4 illustrates this process. In the reminder of this section, we discuss candidate enumeration and query planning. Section 3.4 presents schema optimization and plan recommendation.

3.3.1 Candidate Enumeration

One possible approach to candidate enumeration is to consider all possible column families for a given set of entities. However, the number of possible column families is exponential in the number of attributes, entities, and relationships in the conceptual model. Thus, this approach does not scale well.

Instead, we enumerate candidates using a two-step process based on the application's workload. First, we independently enumerate a set of candidate column families for each query in the application workload. The union of these sets is the initial candidate pool. Second, we supplement this pool with additional column families constructed by combining candidates from the initial pool. The goal of the second step is to add candidates which are likely to be useful for answering more than one query while consuming less space than two

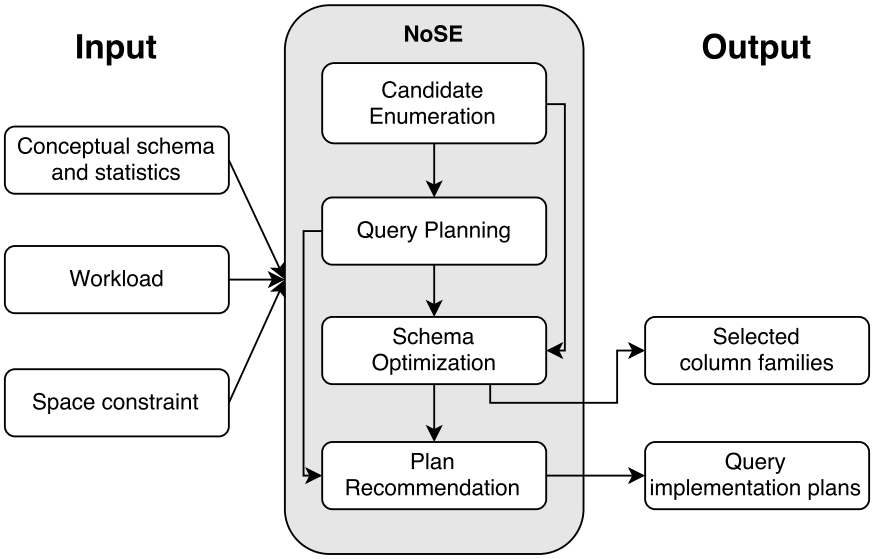


Figure 3.4: Complete schema advisor architecture

separate column families. We do not claim that NoSE's candidate enumerator guarantees the enumeration of column families which result in an optimal schema. However, the optimization process we discuss in Section 3.4 chooses an optimal subset of the enumerated candidates for the given cost model. NoSE's candidate enumerator is pluggable and could be replaced with any enumerator which produces valid column families capable of answering queries in the given workload. We leave other heuristics to determine additional useful column families as future work.

Candidate Column Families

Recall from Section 3.2.3 that a column family is a mapping of the form $\mathcal{K} \mapsto (\mathcal{C} \mapsto \mathcal{V})$. To define a specific column family, we need to determine $\mathcal{K}$, $\mathcal{C}$, and $\mathcal{V}$. That is, we need to specify what the partition keys, clustering keys, and values will be for the column family.

We consider column families in which component of a column family consists of one or more attributes from the application's conceptual model. We represent each column family as a triple, consisting of a set of partition key attributes, an ordered list of clustering key attributes, and a set of value attributes. For example, we can define a column family useful for retrieving, for a given city and state, a list of hotel names, addresses, and phone numbers, in order of hotel name. We represent this column family by the following triple:

`[HotelCity, HotelState][HotelName, HotelID][HotelAddress, HotelPhone]`.

Column families are not limited to containing information on a single entity from the conceptual model. For any query in our language, we can define a column family useful for directly retrieving answers to that query, which we call a *materialized view*. Figure 3.5 describes how NoSE generates materialized views from queries. For example, the query shown in Figure 3.3a, which returns the names and emails of guests who have reserved rooms at hotels in a given city, at room rates above a given rate with a given amenity, corresponds to the following materialized view:

```
[HotelCity, AmenityName]
  [RoomRate, AmenityID, HotelID, RoomID, ResID, GuestID]
  [GuestName, GuestEmail].
```

By supplying city and amenity names, and a minimum room rate, an application can use this column family to retrieve a list of tuples, each of the form `(RoomRate, AmenityID, HotelID, RoomID, ResID, GuestID,GuestName,GuestEmail)`. Each tuple corresponds to a distinct room reservation of a hotel room with the specified amenity and minimum room rate. The query returns tuples in order of `RoomRate`.

Per-Query Candidate Enumeration

The schema optimizer requires flexibility in the choice of column families since its space budget may not allow the recommendation of a materialized view for each query. In addition, when we later consider updates, a column family for each query may become too expensive to maintain. Therefore, in addition to the materialized view, the enumerator also includes additional column families to provide partial answers for each query. The application can use these to answer the query by combining the results of multiple `get` requests to different column families.

To generate the full pool of candidate column families for a given query, we recursively decompose the query at each possible edge in the query graph. Decomposing a query at a specific edge in the query graph splits the query into two parts, which we call the *prefix query* and the *remainder query*. Later, when constructing a query plan, the planner joins these decomposed query graphs along the cut edges to produce a complete plan for a query. Table 3.1 illustrates the first level of this recursive splitting process for the example query from Figure 3.3a. We show the decomposition for just the three labelled edges in (Figure 3.3b).

function: Materialize
input : A query q
output : A materialized view for the query

```
// first entity equality predicates
```
$\mathcal{K} \leftarrow [c.attr \mid c \in q.where \wedge c.op = \texttt{'='} \wedge c.attr.entity = q.from[0]];$

```
// all other equality predicates
```
$\mathcal{C} \leftarrow [c.attr \mid c \in q.where \wedge c.op = \texttt{'='} \wedge c.entity \neq q.from[0]];$

```
// add all other predicates
```
$\mathcal{C} \leftarrow \mathcal{C} + [c.attr \mid c \in q.where \wedge c.attr \notin \mathcal{K} \bigcup \mathcal{C}];$

```
// add ordering attributes
```
$\mathcal{C} \leftarrow \mathcal{C} + (q.order\_by \setminus \mathcal{C});$

```
// end with IDs from all entities
```
$\mathcal{C} \leftarrow \mathcal{C} + [e.id \mid e \in entities(q) \wedge e \notin \{a.entity \mid a \in \mathcal{C}\}];$

```
// selected attributes as values
```
$\mathcal{V} \leftarrow q.select \setminus \mathcal{K} \setminus \mathcal{C};$

return $\mathcal{K} \mapsto (\mathcal{C} \mapsto \mathcal{V});$

Figure 3.5: Materialized view column family generation

Decomposition Edge	Prefix query	Remainder query
1	SELECT Reservation.ResID FROM Reservation.Room.Hotel WHERE Hotel.HotelCity = ? AND Room.Amenity.AmenityName = ? AND Reservation.Room.RoomRate > ?	SELECT Guest.GuestName, Guest.GuestEmail FROM Guest WHERE Guest.Reservation.ResID = ?
2	SELECT Hotel.HotelID FROM Hotel WHERE Hotel.City = ?	SELECT Guest.GuestName, Guest.GuestEmail FROM Guest.Reservation.Room.Hotel WHERE Hotel.HotelID = ? AND Room.RoomRate > ? AND Room.Amenity.AmenityName = ?
3	SELECT Amenity.AmenityID FROM Amenity WHERE Amenity.AmenityName = ?	SELECT Guest.GuestName, Guest.GuestEmail FROM Guest.Reservation.Room.Hotel WHERE Hotel.HotelCity = ? AND Room.Amenity.AmenityID = ? AND Room.RoomRate > ?

Table 3.1: Example of query decomposition for candidate enumeration for the query in Figure 3.3a

For each of the generated prefix/remainder queries, NoSE first enumerates its materialized view. If the `SELECT` clause of the prefix query includes non-key attributes, NoSE enumerates two additional candidates: one that returns only the key attributes, and a second that returns required non-key attributes given the key. For example, for the query in Figure 3.3a, in addition to the materialized view, the enumerator will also generate the following two candidates:

```
[HotelCity, AmenityName]
  [RoomRate, AmenityID, HotelID, RoomID, ResID, GuestID][]

[GuestID][][GuestName,GuestEmail].
```

The former is useful for returning a set of `GuestID`s, given a city, an amenity, and a room rate, and the latter can then produce the guests' names and email addresses.

Finally, the enumerator may generate additional candidates corresponding to *relaxed* versions of the prefix query. Specifically, when the enumerator considers a query of the form

```
SELECT attributes FROM path-prefix WHERE
path.attr op ? AND predicate2 AND …
```

it also generates materialized views for relaxed queries of the form

```
SELECT attributes, attr FROM
path-prefix WHERE predicate2 AND ….
```

That is, the enumerator removes one or more predicates and adds the attributes involved in the predicates to the `SELECT` clause.

Predicates are only considered for removal if the remaining query will have at least one equality predicate remaining. (The application will require this to construct a valid `get` request on the column family in the recommended plan.) NoSE also relaxes queries involving ordering in the same way, by moving an attribute in an `ORDER BY` clause to the `SELECT` list.

The full enumeration algorithm is given in Figure 3.6. For a query with k edges, this algorithm will generate at least $N_k = 1+k+\sum_{j=1}^{k-1} N_j = 2^{k+1}-2$ candidate column families, ignoring any relaxed prefix queries. The number of relaxed prefix queries is exponential in the number of attributes occurring in the `WHERE` and `ORDER BY` clauses, since `Relax(`p`)` considers all subsets of those attributes. Thus, the number of candidates per query grows exponentially with both the size of the query graph and the size of the `WHERE` and `ORDER BY` clauses. However, as we see in Section 3.8.2, this is not a practical concern with the size of several realistic workloads as we show with our runtime analysis in Section 3.8.2.

Candidate Combinations

Once the enumerator has produced candidates for each query in the workload, it then generates additional candidates by combining the per-query candidates that are already present in the pool. Specifically, the enumerator looks for pairs of column families for which both have the same partition key ($\mathcal{K}$), neither has a clustering key ($\mathcal{C}$), and each has different data attributes ($\mathcal{V}$). For each such pair, the enumerator generates an additional candidate that has the same partition keys and all the data attributes from both of the column families it identified. Thus, the new candidate will be larger than either of the original candidates but will be potentially useful for answering more than one query.

function: Enumerate
input : A query q
output : Enumerated column families for q

```
// create a materialized view
```
$C \leftarrow \{\texttt{Materialize}(q)\}$;

foreach *edge* e *in* $q.graph$ **do**

```
    // split prefix and remainder
```
$p, r \leftarrow \texttt{Decompose}(q, e)$;

```
    // add relaxed prefix queries
```
$C \leftarrow C \bigcup \{\texttt{Materialize}(q') \mid q' \in \texttt{Relax}(p)\}$;

```
    // materialize prefix and recurse
```
$C \leftarrow C \bigcup \{\texttt{Materialize(}p\texttt{)}\} \bigcup \texttt{Enumerate(}r\texttt{)}$;

return C;

Figure 3.6: Column family enumeration

`Decompose` splits the query graph in two on the given edge.
`Relax` produces relaxed prefix queries as described in the text.

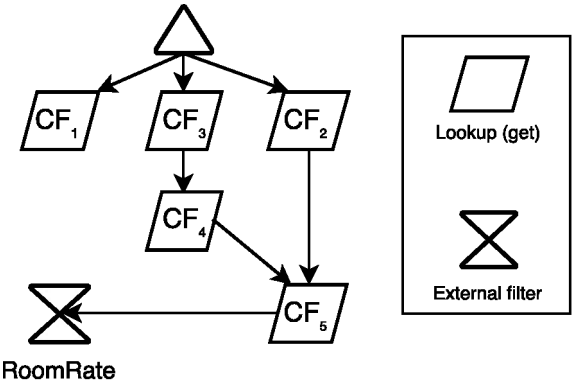
There are additional opportunities for creating new column families by combining candidates from the pool, but NoSE's enumerator currently only exploits this one type of combination. Increasing the number of candidates increases the opportunity for the schema advisor to identify a high-quality schema (i.e. one with lower cost) but this also increases the running time of the advisor. As future work, we intend to explore other opportunities for candidate generation in light of this tradeoff as well as heuristics to prune column families which are unlikely to be useful.

3.3.2 Query Planning

One component of the output of NoSE is a *query execution plan* for each query in the input workload. Query execution plans consist of a series of three possible steps: (a) a `get` or `put` request to the underlying data store, (b) filtering of data fetched from the data store by the application, or (c) sorting of data by the application. Each of these operations has an associated cost, which we describe in Section 3.5. These plans describe to application developers how each query should be implemented.

The task of the query planner is to enumerate all possible plans for evaluating a given query, under the assumption that all candidate column families are available. Each plan is a sequence of steps, using candidate column families, that will produce an answer to an application query. We refer to the result of this process as the *plan space* for the given query. Later, during schema optimization, the schema advisor will use the plan spaces for each query to determine which candidate column families to recommend.

NoSE performs query planning as part of the same recursive decomposition process that generates candidate column families. Consider the decomposition of the running example query (Figure 3.3a) shown in Figure 3.1. For each prefix query, the query planner generates a set of implementation plans, each of which starts by retrieving the results of the prefix query, and finishes by joining those results to a (recursively calculated) plan for the corresponding remainder query. When generating plans for a prefix query, the planner will generate one set of plans for each candidate column family generated for that prefix query, and for any other candidate column families that subsume a candidate for the prefix query. In general, because query planning is based on the same recursive decomposition used to enumerate candidate column families (Figure 3.6), the size of the plan space for a query with k edges in its query graph grows exponentially with k.



CF_1	[HotelCity][AmenityID, RoomRate, RoomID][]
CF_2	[HotelCity][AmenityID, RoomID][]
CF_3	[HotelCity][HotelID][]
CF_4	[HotelID][AmenityID, RoomID][]
CF_5	[RoomID][][RoomRate]

Figure 3.7: Example query plan space

Figure 3.7 shows the plan space for the query below:

```
SELECT Room.RoomID FROM Room WHERE
Room.Hotel.HotelCity = ?city AND
Room.Amenity.AmenityID = ?amenityID AND
Room.RoomRate > ?rate
```

There are three possible plans in the plan space assuming all the enumerated column families are available. The first uses the materialized view CF_1 to answer the query directly. The second finds the `HotelID` for all hotels in a given `HotelCity` using CF_3. The `HotelID` is then used to find all the `RoomID`s for the given hotel using CF_4. Finally, application discovers the `RoomRate`s using CF_5 and filters the `RoomID`s to only contain those matching the predicated on `RoomRate`. The final plan is similar, but goes directly from a `HotelCity` to a list of `RoomID`s using CF_2.

When developing a query plan, it is necessary to select an order in which to execute each individual query and join the values based on the IDs of each entity. Since we require at least one equality predicate for each `get` request, we start with the equality predicate in the query graph with the lowest cardinality. Entities in the query graph are then selected based on those reachable from the currently joined entities and in order by increasing cardinality. While this is not guaranteed to be optimal, choosing entities in this order

reduces the size of intermediate result sets. There are possible alternative heuristics [98] but we use this for simplicity.

3.4 Schema Optimization

A naïve approach to schema optimization is to examine each element in the power set of candidate column families and evaluate the cost of executing each workload query using a plan that involves only the selected candidates. However, this approach scales poorly as it is exponential in the total number of candidate column families.

Papadomanolakis and Ailamaki [103] present a more efficient approach to the related problem of index selection in relational database systems. Their approach formulates the index selection problem as a binary integer program (BIP) which selects an optimal set of indices based on the index configurations that are useful for each query in the workload. Their approach uses a set of decision variables for each query, with the number of variables per query equal to the number of combinations of indices useful to that query. This is still exponential, like the naïve approach, but only in the number of indices relevant to each query, rather than the total number of candidate indices.

Like Papadomanolakis and Ailamaki, we have implemented schema optimization by formulating the problem as a BIP. However, because of the simple structure of the query implementation plans that our schema advisor considers, we are able to provide a simpler formulation for our problem.

Our schema advisor uses the query plan spaces described in Section 3.3.2 to generate a BIP. A binary decision variable, δ_{ij}, exists for each combination of a candidate column family and a workload query. The variable δ_{ij} indicates whether the ith query will use the jth column family in its implementation plan. The objective of the optimization program is to minimize the quantity $\sum_i \sum_j f_i C_{ij} \delta_{ij}$, where C_{ij} represents the cost of using the jth column family in the plan for the ith query and f_i is the query frequency. However, after solving this optimization problem, we run the solver again with an additional constraint that the cost of the workload equals the minimum value which was just discovered, and with the objective of minimizing the total number of column families in the recommended schema. This allows NoSE to produce the schema with the smallest number of column families out of the set of those which are most efficient.

In addition to the decision variables δ_{ij}, our program formulation uses one other decision variable per candidate column family. These variables indicate whether the solution

$$\textbf{minimize} \sum_i \sum_j f_i C_{ij} \delta_{ij}$$

subject to

All used column families being present

$$\delta_{ij} \leq \delta_j, \forall i, j$$

Maximum space usage S

$$\sum_j s_j \delta_j \leq S$$

Plus per-query plan graph constraints (see text)

Figure 3.8: Binary integer program for schema optimization

includes the corresponding column families in the set recommended by the schema advisor. We use δ_j to represent this per-column-family decision variable for the jth candidate column family. Our BIP includes constraints that ensure that

- the solution includes the jth column family in the recommendation if the solution uses it in the plan for at least one query, and
- (optionally) that the total size of the recommended column families is less than the specified space constraint.

To allow sorting to occur at any point in query execution, we also add a constraint that results are properly sorted. Overall, this approach requires $|Q||P|$ variables representing the use of column families in query implementation plans, and $|P|$ variables representing candidate column families, where $|Q|$ represents the number of queries and $|P|$ is the number of candidate column families (Section 3.3.1). We also allow an optional storage constraint whereby the user can specify a limit S on the amount of storage occupied by all column families. The estimated size of each column family s_j is also given as a parameter to the BIP. Figure 3.8 summarizes the binary integer program.

As noted in Figure 3.8, the BIP also requires a set of *plan graph constraints*, on the variables δ_{ij}, which ensure that the solver will choose a set of column families for each query that correspond to one of the plans in the query plan space. These constraints derive from the per-query plan spaces determined by the query planner. For example, in Figure 3.7, the solution can select at most one of CF_1, CF_3, and CF_2 to answer this query,

$$
\begin{aligned}
rl\delta_{1,4} &\leq \delta_{1,3} \\
\delta_{1,5} &\leq \delta_{1,2} + \delta_{1,4} \\
\delta_{1,j} &\leq \delta_j \forall j \in \{1,2,3,4,5\} \\
\delta_{1,1} + \delta_{1,2} + \delta_{1,3} &= 1 \\
\delta_{1,1} + \delta_{1,2} + \delta_{1,4} &= 1 \\
\delta_{1,1} + \delta_{1,5} &= 1
\end{aligned}
$$

Figure 3.9: BIP constraints for the plan graph from Figure 3.7

since each is useful for different plans, and the solution selects only one plan per query. In addition, if the solution selects CF_3, then it must also select CF_4 and CF_5. The BIP will include corresponding constraints on the decision variables δ_{ij} that indicate whether the solution will use those column families to answer this query. Figure 3.9 shows the plan graph constraints for the example shown in Figure 3.7.

After solving the BIP, making the final plan recommendation is straightforward. There is a unique plan with minimal cost based on the values of the decision variables in the BIP.

3.5 Cost Model

The BIP constants C_{ij} represent the cost of using a particular column family in the plan for a particular query. For the example shown in Figure 3.7, there will be five such constants, one for each column family node in the plan graph. As it generates the BIP, NoSE uses its cost model to determine values for these constants.

Each lookup node in the plan graph represents one or more `get` operations against a particular column family. The corresponding BIP constant represents the total cost of all such get operations. NoSE estimates each node's total cost using a two-parameter cost function $T(n, w)$, where n represents the number of `get` operations that the plan will perform against the column family, and w represents the "width" of each request, i.e., the number of $\mathcal{C} \mapsto \mathcal{V}$ pairs that will be returned by each `get`.

NoSE's plans involve only a single `get` from the first column family in each plan. Thus, $n = 1$ for the first column family. The number of `get` operations for the next column family depends on the number of results returned from the `get` on the first column family. Thus,

to estimate values of n for each column family in a plan, NoSE first estimates the result cardinality of the preceding column family — much as in join size estimation in relational databases. For these cardinality estimates, NoSE makes use of simple statistical metadata that is described in terms of the conceptual model and provided as part of NoSE's input. Specifically, a user can provide the cardinality of each attribute in each entity set to NoSE. NoSE also makes use of cardinality constraints from the conceptual model and simple uniformity assumptions. NoSE also uses this same metadata, as well as the properties of the query, to estimate the value of w for each column family. This is a simplistic approach to cardinality estimation. However, cardinality estimation (and costing in general) is not our focus and this approach could easily be replaced by a more sophisticated one.

NoSE's query plans may also include application-side filtering and sorting operations, in addition to column family access. Currently, NoSE's cost module treats filter operations as free. Since the query predicates are simple to evaluate and the application can perform filtering "on the fly" as the underlying record store returns results, filtering adds little to no overhead to the time required to retrieve the records. To account for the cost of application-side sorting, NoSE adds a small constant sorting penalty to the estimated cost of the preceding column family in the plan. A more sophisticated model could adjust this penalty based on result size, and could more accurately account for the overlap of application-side sorting time with retrieval time.

3.5.1 Calibration

The actual cost of performing a plan's `get` operations on a column family depends on the performance characteristics of the underlying extensible record store. Therefore, NoSE learns the cost estimation function ($T(n, w)$) using an offline calibration process.

The calibration process uses a synthetic database consisting of set of column families with differing widths. We performed a series of experiments, by choosing a column family with a particular width w, performing n `get` requests against that column family, and measuring the total time required for the requests. Each request retrieves all w columns for a randomly chosen key. Thus, each such experiment provides a measured execution time for a particular n and w. We performed many of these experiments, and used linear regression over the results to determine $T(n, w)$.

3.6 Updates

The previous sections described how NoSE functions on a read-only workload, but it is important to also consider updates in the workload description. Updates implicitly constrain the amount of denormalization present in the generated schema. This effect results from the maintenance required when the same attribute appears in multiple column families. Each column family containing an attribute modified by an update is also modified, so repetition of attributes increases update cost.

We first introduce extensions to our workload description to express updates. We then describe the update execution plans that NoSE generates and recommends to the application developer. Finally, we describe modifications required to the enumeration algorithm and the BIP used by NoSE to support these updates.

3.6.1 Update Language

In order to support updates to the workload, we extend our query language with additional statements which describe updates to data in terms of the conceptual model, as illustrated in Figure 3.10. `INSERT` statements create new entities and result in insertions to column families containing attributes from that entity. We assume that the `INSERT` statement provides the primary key of each entity, but all other attributes are optional. `UPDATE` statements modify attributes, resulting in updates to any corresponding column families in the schema. `DELETE` statements remove all data about deleted entities from any associated column families. Both `UPDATE` and `DELETE` statements specify the entities to modify using the same predicates available for queries. Finally, `CONNECT` and `DISCONNECT` statements create or destroy relationships between entities. These statements simply specify the primary key of each entity and the relationship to modify. We also allow the creation of relationships on the insertion of a new entity by specifying foreign keys of related entities.

3.6.2 Update Plans

As with queries, NoSE must provide an implementation plan for each update, using the `get`, `put` and `delete` operations supported by the extensible record store. Because NoSE may denormalize attributes across multiple column families, it must first determine which column families are affected by the update, and then generate a plan for modifying each of those column families to reflect the changes.

```
INSERT INTO Reservation SET ResID = ?, ResStartDate = ?, ResEndDate = ?

DELETE FROM Guest WHERE Guest.GuestID = ?guestID

UPDATE Reservation FROM Reservation.Guest SET Reservation.ResEndDate = ?
  WHERE Guest.GuestID = ?guestID

CONNECT User(?userID) TO Reservations(?resID)

DISCONNECT User(?userID) FROM Reservations(?resID)
```

Figure 3.10: Example NoSE update statements

Query

```
SELECT Room.RoomRate FROM Room.Hotel.PointsOfInterest
WHERE Room.RoomFloor = ?floor AND PointsOfInterest.POIID = ?poiID
```

Materialized View Column Family

```
[Room.RoomFloor][PointsOfInterest.POIID, Hotel.HotelID, Room.RoomID]
    [Room.RoomRate]
```

Update

```
UPDATE Room FROM Room.Reservations.Guest SET RoomRate = ?rate1 WHERE
Guest.GuestID = ?id AND Room.RoomRate = ?rate2
```

Figure 3.11: An example NoSE query, materialized view, and update

In general, a NoSE update statement might not contain enough information to allow NoSE to construct an update plan. To illustrate this problem, suppose that the schema recommended by NoSE includes the materialized view shown in Figure 3.11. Suppose further that the workload includes the `UPDATE` shown in the figure. Such an `UPDATE` may affect the materialized view since the `UPDATE` changes the room rates stored in the view. Thus, the recommended plan for this update should include making changes to the materialized view, using `put` or `delete` operations. However, a plan cannot change room rates in this view without knowing the `RoomFloor`, `POIID`, `HotelID`, and `RoomID` associated with the room rates that need to change. This information is not provided by the `UPDATE`.

To resolve this problem, NoSE update plans may include *support queries*, which obtain information that the plan requires in order to perform the update. NoSE generates such support queries automatically, as needed, as part of its planning and optimization process. Given an update from the workload and candidate column family, Figure 3.12 describes how NoSE generates the support queries necessary to update the column family. If an update includes all the information required to update a column family, then Figure 3.12 does not generate any support queries.

Plans for `UPDATE`

We will use the materialized view and `UPDATE` from Figure 3.11 to illustrate how NoSE generates plans for updates. For this `UPDATE` and view, NoSE will generate two support queries. The first will return `RoomID`s for the rooms whose rates the `UPDATE` modifies. NoSE executes the second query once for each `RoomID` returned by the first query. The query returns the `RoomFloor`, `HotelID`, and `POIID`s for the given `RoomID`. Figure 3.13 illustrates these two support queries. The results of this second query identify the record keys (`RoomFloor`s) and columns the `UPDATE` needs to modify `RoomRate` in the materialized view. The plan recommended by NoSE can then perform each update using a `put` command against the materialized view.

In general, NoSE may require more than two support queries to update a column family, although two is sufficient in our example. The number of support queries that NoSE generates depends on the structure of the materialized view's query graph, and on which entity NoSE is updating. Furthermore, applying the updates to the column family once the necessary records and columns have been identified is not always as simple as our example suggests. If an `UPDATE` modifies an attribute used as part of the column names or part of the record key in the view, then NoSE cannot simply `put` the new value as it does in our previous example. Instead, such updates delete the old record or column and then insert a replacement.

function: Support
input : A modification u and a column family t
output : A set of support queries

`// Get required attributes`
$A \leftarrow$ `Required`(t, u);
if $|A| == 0$ **then return** $\emptyset$;

if $u.type \in \{$*Insert*, *Connect*, *Disconnect*$\}$ **then**
    `// Split on relevant edges`
    $E \leftarrow c \mid c \in u.connections \wedge c \in t.graph$; $G =$ `Split`$(t.graph, E)$;
    `// Build a query for each subgraph`
    $S \leftarrow \emptyset$;
    foreach *subgraph* g *in* G **do**
        $A' \leftarrow \{a \mid a \in A \wedge a.entity \in g\}$; $w' \leftarrow \{c \mid c.attr.entity \in g\}$;
        if $|A'| = 0$ **then continue**;
        $S \leftarrow S \bigcup \{(u.from, A', w', [])\}$;
    return S;
else return $\{(u.from, A, u.where, [])\}$;

`Required` produces the necessary fields for the update. `Split` splits a graph on a given edge.

Figure 3.12: Support query generation

```
SELECT Room.RoomID FROM Room.Reservations.Guest
WHERE Guest.GuestID = ?id AND Room.RoomRate = ?rate2

SELECT Room.RoomFloor, Room.Hotel.HotelID, PointsOfInterest.POIID
FROM PointsOfInterest.Hotels.Rooms WHERE Room.RoomID = ?id
```

Figure 3.13: Support queries for the update shown in Figure 3.11

Plans for `DELETE`

Plans for `DELETE` statements are similar to those for `UPDATE`s, since both types of statement affect a single type of entity in the conceptual model. Like an `UPDATE`, a `DELETE` may require support queries to determine which records to remove from an affected column family.

Plans for `INSERT`

If an `INSERT` statement does not include any `CONNECT` clauses, then NoSE only needs to update column families that contain attributes of the newly inserted entity. In this case, the `INSERT` supplies all the necessary attribute values, and NoSE does not need to generate any support queries.

If an `INSERT` does include `CONNECT` clauses, then NoSE may need to update column families that include attributes from multiple types of entities, including the type of the inserted entity. Since the `INSERT` statement only specifies values for the attributes of the inserted entity, NoSE will construct one or more support queries to obtain the attribute values for other entities that appear in the column family. Furthermore, since the new entity's attribute values may be denormalized in the column family, the support queries determine how many new records or columns to add to the column family to reflect the addition of the new entity. For example, an `INSERT` of a new POI, with a `CONNECT` to a nearby hotel, may result in the addition of multiple columns in multiple records of the materialized view shown in Figure 3.11. Specifically, there will be a new column for each room in the hotel linked to the new POI. Support queries determine the `RoomID`, `RoomFloor`, and `RoomRate` of these rooms, so that the `INSERT` plan can add the necessary columns to the column family.

Plans for `CONNECT` and `DISCONNECT`

`CONNECT` and `DISCONNECT` may modify a column family if the column family's underlying query graph includes the edge that is being connected or disconnected. `CONNECT` statements may cause new records or columns to be inserted, and support queries obtain the necessary attribute values, much as was done for `INSERT`. Similarly, `DISCONNECT` may cause records or columns to be removed, and support queries determine the affected records and columns. When modeling the cost of `CONNECT` and `DISCONNECT`, we treat these as insertions or deletions to column families involving the relevant edge.

3.6.3 Column family enumeration for updates

Additional column families may be necessary to answer support queries for updates in the workload. When there are updates, the candidate enumerator uses the procedure shown in Figure 3.14. This procedure extends the candidate enumeration procedure for query-only workloads, which was originally described in Section 3.3.1. As shown in Figure 3.14,

function: UpdateEnumerate
input : A set of queries Q and updates U
output : A set of enumerated column families

```
// enumeration for workload queries
C ← {Enumerate(q) | q ∈ Q};
// enumeration for support queries
do twice
    C' ← C;
    foreach update u in U do
        foreach column family c in C' do
            if Modifies?(u, c) then
                foreach query q in Support(u, c) do
                    C ← C ⋃ Enumerate(q);
return C ⋃ Combine(C);
```

The `Enumerate` and `Combine` functions represent the candidate enumeration and candidate combination methods for query-only workloads from Section 3.3.1. `Support` is the support query generation algorithm (Figure 3.12). `Modifies?` tests whether an update requires modifications to a given column family.

Figure 3.14: Column family enumeration for workloads with updates

NoSE performs candidate enumeration for each query in the original workload, and twice for support queries. This is because support queries generated on the first iteration may cover new edges in the entity graph. Candidate column families for these support queries may themselves be affected by workload updates, resulting in support queries for support queries.

Update support queries increase the size of the application workload. For example, suppose that the original workload includes a query Q with a query graph of length k, and an update U that affects the first entity in Q's query graph. This will result in the addition of at least $k + 1$ support queries to the workload. This is because candidate enumeration for Q will generate $k + 1$ prefix queries involving the updated entity. Each of those queries will have a materialized view which is affected by U, and for which a support query will be required.

$$\textbf{minimize} \sum_i \sum_j f_i C_{ij} \delta_{ij} + \sum_m \sum_n f_m C'_{mn} \delta_n$$

subject to

All constraints from Figure 3.8

Additional constraints per support query (see text)

Figure 3.15: BIP modifications for updates

3.6.4 BIP Modifications

To incorporate updates into our BIP, we first add constraints for all support queries similarly to those for queries in the original workload. In addition, we add constraints to ensure that NoSE does not generate plans for support queries for a candidate column family unless that column family is part of the recommended design. The objective function receives an additional term, $\sum_m \sum_n f_m C'_{mn} \delta_n$, to represent the cost of updating each column family, which is contingent on the recommendation including this column family in the final schema. C'_{mn} is the cost of updating column family n for update m (with frequency f_m) given that the column family appears in the final schema (δ_n). The cost of support queries is also added using the same weight specified for the update in the workload. Figure 3.15 shows the modified BIP.

After solving this modified BIP, NoSE plans each update by first generating any necessary support query plans in the same way as plans for queries in the original workload. Each update plan then consists of a series of support query plans along with insertion or deletion as necessary for the update.

3.7 Case Study

In this section, we present an analysis of a partial workload extracted from EasyAntiCheat (EAC)[1], a real-time cheat detection engine for multiplayer games. Our goals are to illustrate the challenges of NoSQL schema design and to illustrate how NoSE works. In Section 3.8, we present a more quantitative evaluation of NoSE.

[1] www.easyanticheat.net

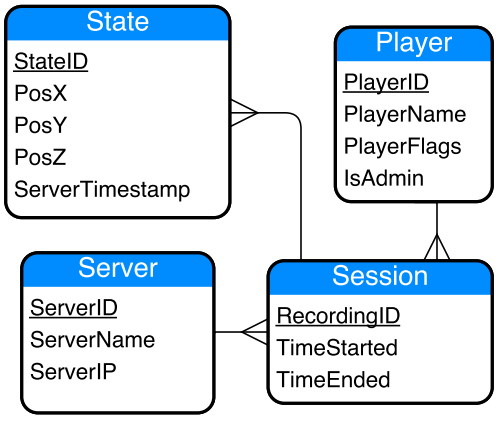


Figure 3.16: Entities modeled in EasyAntiCheat

EAC receives large volumes of player behaviour data in real time. Their backend systems pull in this data and analyze player behaviour to determine patterns indicative of cheating. After hitting scalability limits with their relational database infrastructure, EAC considered Cassandra as a possible backend. Figure 3.16 shows a simplified version of the conceptual model for the application. Game servers have a number of player sessions with servers continually collecting information on the state of players in each session. The system stores data on millions of players and states, hundreds of thousands of player sessions and thousands of servers. Players generate new states at rates of up to several hundred thousand per second.

For this case study, we focus on a subset of the workload. Figure 3.17 shows the most important queries in the workload. The workload also includes updates (not shown in Figure 3.17) including the insertion of new player states, sessions, players, and servers. EAC estimates their workload to be roughly 80% writes and 20% reads. The majority of the writes come from the insertion of new states while most queries are instances of Q_1 and Q_2. We have assumed specific frequencies, fitting these constraints, for all queries and updates. For simplicity, we assume these are the only queries and updates performed by the system.

We used the EAC schema and workload as input to NoSE, and Figure 3.18 shows the five recommended column families. The critical problem NoSE must resolve for this workload is how to store player states, which are voluminous and frequently inserted, and which are read by both Q_1 and Q_2. Furthermore, one of these queries retrieves states for a single player session, while the second retrieves states for all players on a game server. NoSE addresses this problem by recommending a single column family (CF_1) to support

Q_1 Get the latest state of a player

```
SELECT states.PosX, states.PosY, states.PosZ, states.ServerTimestamp
FROM Server.sessions.states WHERE Server.ServerID = ?
AND sessions.player.PlayerID = ? ORDER BY states.ServerTimestamp
```

Q_2 Get the latest states for all players on a server

```
SELECT states.PosX, states.PosY, states.PosZ, states.ServerTimestamp,
sessions.player.PlayerID FROM Server.sessions.states WHERE
sessions.player.IsAdmin = 0 AND Server.ServerID = ?
AND states.ServerTimestamp > ? AND states.ServerTimestamp <= ?
ORDER BY states.ServerTimestamp
```

Q_3 Get information on an individual server

```
SELECT Server.ServerName, Server.ServerIP
FROM Server WHERE Server.ServerID = ?
```

Q_4 Check if a server exists

```
SELECT Server.ServerID FROM Server WHERE Server.ServerID = ?
```

Q_5 Get sessions for a player

```
SELECT Session.SessionID FROM Session.player WHERE player.PlayerID = ?
```

Figure 3.17: Important queries in the EasyAntiCheat workload

```
CF1 [Server.ServerID]
  [PlayerState.ServerTimestamp, Player.PlayerID, PlayerState.StateID,
    Session.SessionID][Player.IsAdmin,
      PlayerState.PosX, PlayerState.PosY, PlayerState.PosZ]

CF2 [Server.ServerID][][Server.ServerName, Server.ServerIP]

CF3 [Player.PlayerID][Session.SessionID][]

CF4 [Session.SessionID][Player.PlayerID, Server.ServerID, Player.IsAdmin][]

CF5 [Player.PlayerID][][Player.IsAdmin]
```

Figure 3.18: Column families produced for the EasyAntiCheat workload

both queries, so that state information is not denormalized and new states are only inserted in one place. It chooses an organization for CF_1 that can support both Q_1 and Q_2, with the help of some application-side processing. Specifically, the schema partitions states according to the game server they originate from and sorts them by timestamp. This allows the extensible record store to directly support the timestamp range predicates in Q_2, and allows the application to avoid sorting by retrieving states in timestamp order. NoSE's recommended plan for Q_1 is a single `get` from CF_1 followed by application-side filtering on `PlayerID`. Similarly, Q_2's plan is a single `get` followed by filtering on `IsAdmin`. To support this filtering, NoSE has denormalized players' `IsAdmin` attributes into CF_1 to avoid the need for additional lookups to retrieve that information.

Column families CF_2 (Q_3) and CF_3 (Q_4 and Q_5) provide answers to the remaining queries, which are less frequent. Column families CF_4 and CF_5 are examples of column families that provide answers to support queries for updates, as discussed in Section 3.6. To insert a new player state from a given session into CF_1, the application must also know the player and server associated with that session, as well as the player's `IsAdmin` value, since that information is denormalized into CF_1 to support queries. Thus, NoSE's plan for insertions of new player states is a `get` from CF_4 to obtain the necessary player and server information for the new state's session, followed by a `put` of a new record into CF_1. Similarly, when a new session in created, NoSE's plan first obtains the `IsAdmin` value for the session's player from CF_5 before inserting the new session into CF_4.

NoSE's schema recommendations are sensitive to workload and database properties, such as the relative frequencies of the various queries and updates and the entity cardinalities. For example, CF_1 may become a poor way to support Q_1 if the number of players per game server gets too large. On the one hand, this sensitivity is a positive, as it reflects the reality of the underlying NoSQL systems, and it allows an application developer to explore the schema design space using NoSE, by simply tweaking workload parameters. On the other hand, it suggests an interesting direction for future work, which is the recommendation of schemas with performance that is robust across a range of workload changes, though not necessarily optimal at any point within the range.

3.8 Evaluation

In this section we present an evaluation of NoSE designed to address two questions. First, does NoSE produce good schemas (Section 3.8.1)? Second, how long does it take for NoSE to generate schema recommendations (Section 3.8.2)? NoSE is available on GitHub [97].

3.8.1 Schema Quality

To evaluate the schemas recommended by NoSE, we used it to generate schema and plan recommendations for a target application. We then implemented the recommended schema in Cassandra along with the recommended application plans. While executing the plans against Cassandra, we measured their execution times. Similarly, we also implemented and executed the same workload against two baseline schemas for comparison. A full description of the workload and the associated schemas is provided in Appendix A.

Although extensible record stores like Cassandra are in wide use, we are not aware of open-source applications or benchmarks. One exception is YCSB [43], which is useful for performance and scalability testing, but offers no flexibility in schema design. Instead, we created a target application by adapting RUBiS [30], a Web application benchmark originally backed by a relational database which simulates an online auction website.

To adapt RUBiS for Cassandra, we created a conceptual model based on the entities managed by RUBiS. The resulting model contains seven entity sets, with ten relationships among them. Using this model, we generated a NoSE workload description, with queries and updates weighted according to the bidding workload defined by RUBiS. This workload consists of one or more statements corresponding to each SQL statement used in the original RUBiS workload.

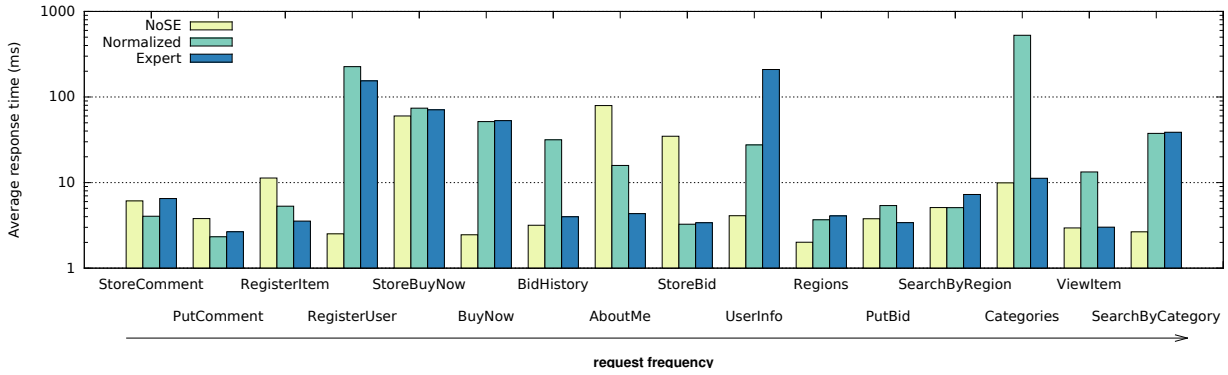


Figure 3.19: Response time of RUBiS request types using three different schemas.

Request types are ordered from least frequent to most frequent.

The first schema we examine, the *NoSE schema*, was recommended by NoSE using no storage constraint. We chose not to evaluate the effect of storage constraints since updates have a similar affect of reducing the amount of denormalization in the resulting schema. This results in a highly denormalized, workload-specific schema, generally consistent with the rules of thumb for NoSQL schema design. We compared this to two baseline schemas. The first, which we refer to as the *Normalized* schema, is a manually created schema which is highly normalized. For each entity set, it includes a column family for which the partition key is the primary key of the entity and which stores all data associated with the entity. The Normalized schema also includes column families which serve as secondary indices for queries which do not specify entity primary keys. These column families use the attributes given in query predicates as the partition keys and store the primary key of the corresponding entities. A human designer who is familiar with Cassandra defined the second baseline, which we refer to as the *Expert* schema, using the same workload that was input to NoSE. The expert schema's designer also defined an execution plan for each query. The Normalized schema is ∼4.9GB on disk compared to ∼6.7GB for the expert schema and ∼7.8GB for the schema produced by NoSE. Details of each schema and execution plan are provided in Appendix A.

We implemented each schema in Cassandra, and populated each with data for a RUBiS instance with 200,000 users. Rather than building a custom application to target each schema, we developed a client-side execution engine which can interpret and execute the query plans specified in the plan format used by NoSE. This engine executed the plans created for all three schemas. Query and update plans for the two baselines were manually developed, and the NoSE schema uses plans recommended by NoSE.

We used two servers for each experiment, one to execute the client-side query plans and one running an instance of Cassandra 2.0.9. Each server has two six core Xeon E5-2620 processors operating at 2.10 GHz and 64 GB of memory. A 7200 RPM SATA drive stored the Cassandra data directory. The experiments ran with all Cassandra-level caching disabled since we do not attempt to model the effects of caching in our cost model. NoSE could incorporate a more elaborate cost model which captures the effects of caching, as its cost model is pluggable.

The RUBiS workload consists of sixteen types of application-level requests called *interactions*, each implemented using one or more queries or updates against the underlying database. Figure 3.19 shows the mean response time for requests of each type, for each of the schemas that we evaluated. Mean response times for the different types ranged from 2.0–79.5 ms for the schema recommended by NoSE, 1.3–526.8 ms for the Normalized schema, and 2.7–209.4 ms for the Expert schema. The weighted overall average response times (over all request types) were 8.4ms, 87.0ms, and 41.6ms for the NoSE, Normalized, and Expert schemas, respectively. Thus, NoSE's schema results in speed-ups of 10.2× and 4.9× relative to the two baselines. Performance for individual requests using the NoSE schema was not better than that of the baselines for all request types. However, overall performance improves because NoSE's cost-based optimizer allows it to exploit workload information to provide good performance for the most frequent operations (those on the right in Figure 3.19). In particular, the NoSE schema uses extensive denormalization to support fast execution of frequent queries, at the expense of additional work during (less frequent) updates.

In addition to the experiment shown in Figure 3.19, we also experimented with variations of the RUBiS workload that have different mixes of request frequencies. Figure 3.20 shows the results, for four different mixes arranged in order of increasing write intensity. We also considered RUBiS's *Browsing* workload mix and two variations of the Bidding mix with the relative frequency of update interactions increased by factors of 10 and 100 relative to the original Bidding workload. The Browsing workload consists of 7 read-only interactions and the Bidding workload adds 9 interactions involving updates. The total frequency is approximately 23% update interactions for the Bidding workload. For each mix, NoSE generated a schema and execution plans specific to that mix, which we compared against the original Expert and Normalized schemas. Note that we only expect the Expert schema to perform well against the Bidding workload used for its development.

NoSE is extremely effective on the Browsing mix since it is free to denormalize heavily with no update penalty. As the workload becomes more write intensive, NoSE's schema recommendations become more normalized, and workload performance approaches that achieved by the Normalized baseline. In the most update intensive mix, NoSE's schema

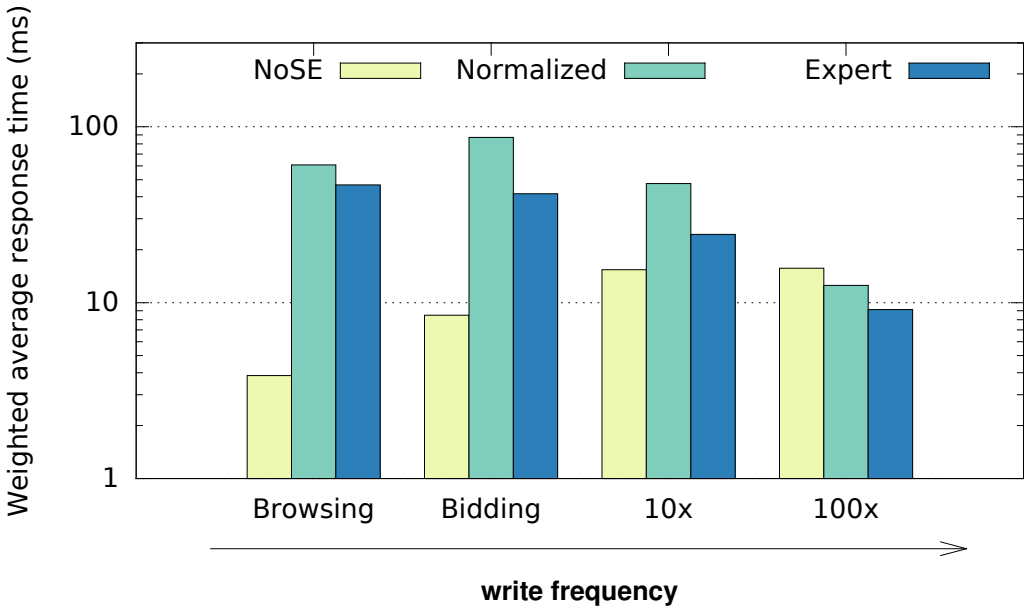


Figure 3.20: Execution plan performance for different request mixes.

10× and 100× refer to the Bidding workload with update request frequencies increased by 10× and 100×.

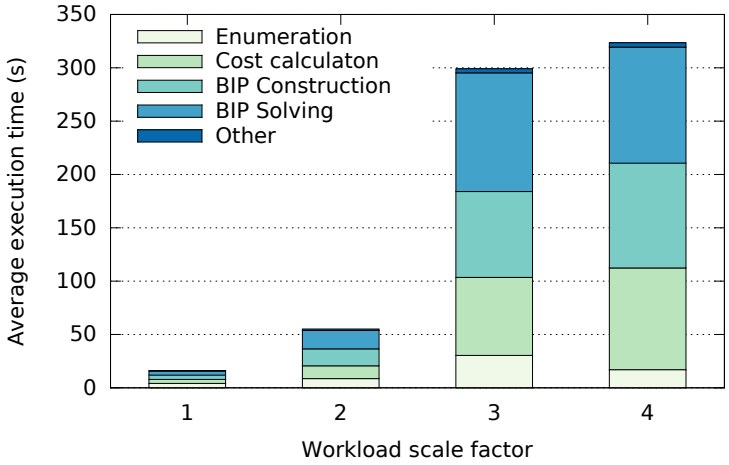


Figure 3.21: Advisor runtime for varying workload scale factors

performs slightly worse than the baseline schemas. NoSE has no knowledge of the correlation of queries in the input workload and cannot share the results of support query execution. In contrast, the expert schema does exploit this knowledge and is thus able to avoid unnecessary queries.

3.8.2 Advisor Runtime

Running NoSE for the RUBiS workload takes approximately 3.3 minutes. To evaluate the advisor runtime for workloads larger than RUBiS, we generated random entity graphs and queries to use as input to our tool. The entity graph generation uses the Watts-Strogatz random graph model [127]. After generating the graph, we randomly assign a direction to each edge and create a foreign key at the head node. We then add a random number of attributes to each entity in the graph. Our generator uses a random walk through the graph to identify the graph of each statement. For any statements involving a `WHERE` clause, we randomly generate predicates in the graph. Queries and updates select or update randomly chosen attributes in the graph.

Figure 3.21 shows the results of a simple experiment in which we started with a random workload having similar properties to the RUBiS workload discussed in the previous section. We then increased the size of the workload by multiplying the number of entities and statements by a constant factor. The largest workload (4× scale factor) tested contains 120 queries, 12 updates, and 20 insertions over an entity graph with 28 entities. The

figure shows the time required for NoSE to recommend a schema and a set of execution plans as a function of this factor. We ran all experiments using a machine with the same specifications as in the previous section. The increase in runtime is a result of the increased number column families enumerated, which also increases the number of support queries NoSE considers. This interaction increases non-linearly with the workload size (e.g., increased numbers of queries and updates) since there are more ways that column families recommended for queries interact with updates. There is likely room for optimization in the NoSE code to significantly reduce the runtime. For example, any heuristics which can exclude column families from enumeration will reduce the runtime at all further stages in the process.

3.9 Summary of NoSE and Future Directions

Schema design for NoSQL databases is a complex problem with additional challenges as compared to the analogous problem for relational databases. We have developed a workload-driven approach for schema design for extensible record stores, which is able to effectively explore tradeoffs in the design space. Our approach implicitly captures best practices in NoSQL schema design without relying on general design rules-of-thumb, and is thereby able to generate effective NoSQL schema designs. Our approach also allows applications to explicitly control the tradeoff between normalization and query performance by varying a space constraint.

Currently, NoSE only targets Cassandra. However, we believe that with minimal effort, the same approach could apply to other extensible record stores, such as HBase. We also intend to explore the use of similar techniques for data stores with different data models, such as key-value stores and document stores. Applying our approach to such data stores may only require changing the cost model and the physical representation of column families.

The same triple notation used to represent column families for Cassandra may also be applicable for representing structures in other data models. For example, we have experimented with adding support for the MongoDB [8] document store to NoSE. Suppose we have the following triple constructed to store data for an application using the entity graph in Figure 3.1:

```
[GuestID,ResID,RoomID][]
  [GuestName,GuestEmail,ResStartDate,ResEndDate]
```

In this case, we can use the structure of the related entities in the entity graph to define levels of nesting in a document collection. One example of this is a document like the one below which has `GuestID` values as keys for each document. Reservations and rooms are nested within each document. Since there is a one-to-many relationship between guests and reservations, we can represent the reservations as an array inside the document. Each reservation only has a single room, so the `RoomID` for each reservation can simply be stored as a nested property.

```
{ "Guest1": {
  "GuestName": "Alice", "GuestEmail": "alice@abc.com",
  "Reservations: [
  {"Res1": {"ResStartDate": "2018/08/21", "ResEndDate": "2018/08/24",
            "Room": {"RoomID": "Room1"}}}
  ]}}
```

More investigation is required to consider what the structure of documents should look like in the general case. In addition to this consideration, we believe NoSE may require more significant changes to fully exploit capabilities present in different NoSQL databases such as secondary indexes or more expressive query languages.

Chapter 4

ESON: Schema Recovery from Denormalized Physical Designs

As discussed in the previous chapter, although NoSQL systems may not require applications to define rigid schemas, application developers must still decide how to store information in the database. These choices can have a significant impact on application performance as well as the readability of application code [61]. For example, consider an application using HBase to track requests to an on-line service. To store records of requests in an HBase table, the application must decide how to represent the requests. Should there be a record for each request, or perhaps one record for all the requests from a single client? What column families will be present in the table, and what will they represent? The same requests may be stored in multiple tables since the structure of tables determines which queries can be asked. The choice of data representation depends on how the application expects to use the table, i.e., what kinds of queries and updates it needs to perform. These kinds of decisions are automated with NoSE, but many existing applications have manually designed schemas. In this case, since the NoSQL system itself is unaware of any schema design decisions, it can provide little to no help in understanding what is being represented in the database.

In our on-line service example, request information might be stored twice, once grouped and keyed by the customer that submitted the request, and a second time keyed by the request subject or the request time. If the application updates a request, or changes the information it tracks for each request, these changes should be reflected in both locations. As discussed in Chapter 1.3, this denormalization (duplication of data) is done by the application developer. The database system is unaware of this denormalization and unable to help manage updates and queries over this denormalized data. We aim to recover

explicit knowledge of application-level denormalization through a task we refer to as *schema renormalization*. This chapter addresses the schema renormalization problem through the following technical contributions:

- We present a complete semi-automatic technique for extracting a normalized conceptual schema from an existing denormalized NoSQL database. Our technique works with different types of NoSQL systems (e.g., Cassandra, MongoDB) through a common unifying relational representation of the physical structures in the NoSQL database. It produces a normalized conceptual schema for the database, such as the one represented graphically as an entity-relationship diagram, in Figure 4.1. In addition, it produces a *mapping* from each NoSQL physical structure to the conceptual model. Connecting the physical and conceptual schemas this way increases their utility, as discussed in Section 4.6.

- We develop an automatic normalization algorithm (Section 4.4), which forms the core of our semi-automatic approach. This algorithm uses both functional and inclusion dependencies to extract the conceptual model from the NoSQL system's physical structures. Our algorithm, which we call ESON, ensures that the resulting model is in *interaction-free inclusion dependency normal form*, indicating the redundancy implied by the input dependencies (both functional and inclusion) has been removed from the schema. To the best of our knowledge, this is the first normalization algorithm which does this.

- Our normalization algorithm requires functional and inclusion dependencies as input. These may be provided by a knowledgeable user, or may be mined from an instance of a NoSQL database. Our third contribution is a technique for adapting existing relational dependency mining techniques to provide the dependencies required by the normalization algorithm (Section 4.5).

- Finally, in Section 4.7, we present a series of case studies which show the full schema renormalization process in action for several NoSQL applications. We use these case studies to highlight both the advantages and the limitations of our approach to renormalization.

A short paper based on the work in this chapter has been accepted for presentation at the International Conference on Conceptual Modeling (ER 2018) [94].

The conceptual data model that our algorithm produces can serve as a simple reference, or specification, of the information that has been denormalized across the workload-tuned

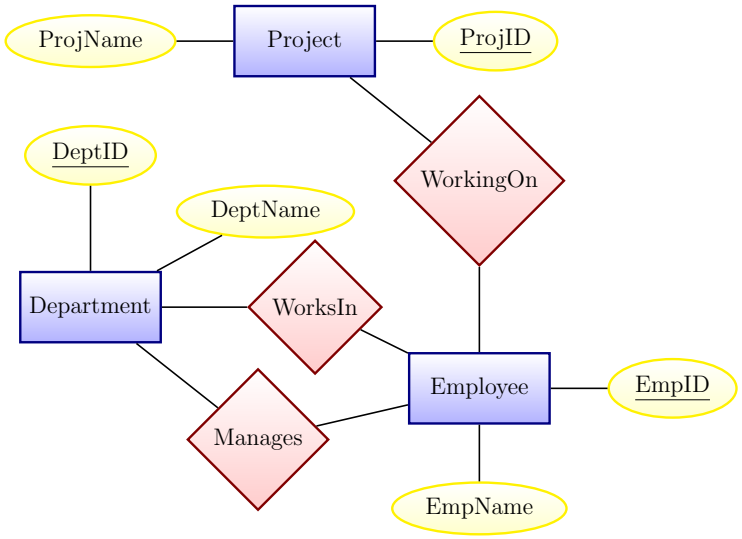


Figure 4.1: Schema example after renormalization

physical database structures. We view this model as a key component in a broader methodology for schema management for NoSQL applications. Current processes for managing schema evolution in NoSQL datastores are entirely manual and error prone. If the application workload changes, or if new types of data are added to the database, a developer must evaluate how the physical schema must evolve to support the change. This process must be repeated each time the application evolves, with the potential for errors each time.

We would like to support an alternative methodology for schema evolution, which is illustrated in Figure 4.2. The first step is construction of a normalized conceptual data model over the denormalized schema. This is the problem we address in this thesis. This conceptual model enables several new use cases. Through the knowledge of the denormalization present in the existing physical structures, we can determine efficient plans for executing ad-hoc queries over the denormalized data. In addition, the application developer then can "lift" existing applications by describing the application's existing queries and updates against the conceptual model. Instead of directly changing the physical schema, application developers can then evolve the application at the level of the conceptual model, e.g., by adding additional data, or by adding or modifying (conceptual) queries. Once the application has been evolved at the conceptual level, existing schema design tools and techniques, such as NoSE and NoAM [14], can then be used to generate a new, workload-aware, denormalized physical database design for the target NoSQL system.

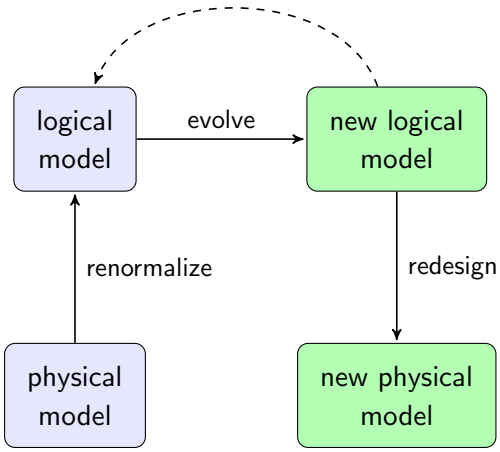


Figure 4.2: NoSQL schema evolution lifecycle

4.1 Renormalization Overview

We renormalize NoSQL databases using a three step process. The first step is to produce a *generic physical schema* that describes the physical structures that are present in the NoSQL database. This step may be performed manually, although some tools exist to aid in automation, which we discuss later. The generic physical schema serves to abstract differences among the database models of different types of NoSQL systems. For example, HBase uses tables with one or more column families, while MongoDB stores collections of JSON documents. The generic physical schema hides these differences, providing a uniform way to represent the physical structures that are present in the NoSQL store. It does not capture all of the characteristics of these structures. In particular, it does not capture how they can be used by applications, and it does not capture features of the structures that affect performance. Rather, it focuses on describing the information that is present in these structures, which is what is needed for renormalization. We describe the generic physical model in more detail in Section 4.2, and illustrate how it can be produced for different types of NoSQL systems.

The second step in the renormalization process is to identify dependencies among the attributes of the generic physical model. The required dependencies can be provided by a user with understanding of the NoSQL system's application domain [91] or automatically using existing dependency mining techniques, which we explore in Section 4.5. We discuss the required dependencies further in Section 4.3.

The final step in the renormalization process is to normalize the generic physical schema using the dependencies, resulting in a logical schema such as the one represented (as an ER diagram) in Figure 4.1. This step is automated, using the procedure described in Section 4.4. Our algorithm ensures that the normalized schema is in *inclusion dependency normal form (IDNF)* which informally means that redundancy in the physical schema captured by the provided functional and inclusion dependencies is removed.

Although we do not discuss this further in this thesis, it is also possible to apply this three-step methodology iteratively, to incrementally renormalize a database. In particular, one can start with a *partial* physical schema (e.g. a single table), renormalize it, and then gradually add to the schema and renormalize until the full physical schema has been renormalized.

4.2 The Generic Physical Schema

The first step in the renormalization process is to describe the NoSQL database using a generic schema. The schemas we use are relational. Specifically, a generic physical schema consists of a set of (flat) *relation schemas*. Each relation schema describes a physical structure in the underlying NoSQL database. A relation schema, in turn, consists of a unique relation name plus a fixed set of attribute names. Attribute names are unique within each relation schema.

The procedure for doing this depends on the type of NoSQL database that is being normalized. Here, we illustrate the process using examples based on three different types of systems: Cassandra, HBase, and MongoDB. Our examples are based on RUBiS, an online auction application which we describe in more detail later, in Section 4.7.1.

Cassandra: NoSQL systems differ in the amount of schema information that they understand. In Cassandra, data is stored in tables, which applications can define using CQL, an SQL-like language. CQL includes a `CREATE TABLE` statement, which allows the application to define the structure of a table. Figure 4.3 shows an example of a CQL definition of a single table from the RUBiS database. This table records the IDs of bids for each item under auction. Information about the bids, such as the bid date, is denormalized into this table so that an application can retrieve it without having to perform joins, which Cassandra does not support.

If the NoSQL database includes a well-defined schema, as in this example, then describing the physical schema required for renormalization is a trivial task. Figure 4.3 also

CQL

```
CREATE TABLE ItemBids(itemID uuid, bid decimal,
bidID uuid, quantity int, date timestamp,
PRIMARY KEY(itemID,bid,bidID));
```

Generic schema

```
ItemBids(itemID, bid, bidID, quantity, date)
```

Figure 4.3: A CQL **ItemBids** table, and corresponding schema

2315	bids:25,b9734 16,2016-05-02	bids:24,b3267 6,2016-05-02	bids:22,b9907 8,2016-05-01
2416	bids:65,b7633 1,2016-04-09	bids:60,b9028 ,2016-04-01	

Figure 4.4: An **ItemBids** table in HBase

shows the generic relation schema for the CQL **ItemBids** table. The generic schema simply identifies the attributes that are present in the table, and gives names to both the attributes and the table itself. In the case of Cassandra, these names can be taken directly from the CQL table definition. The `PRIMARY KEY` declaration in the CQL table definition also provides information about functional dependencies among the tables attributes. We defer the further discussion of these dependencies to Section 4.3.

HBase: Like Cassandra, HBase stores data in tables. Each table contains one or more column families. However, HBase understands only table names and the names of the tables' column families. Individual columns in each column family are not fixed. Different rows in the same table may have different columns. Figure 4.4 shows two rows from an HBase table that stores the same information (about bids for each item) that a Cassandra application would store in the table from Figure 4.3. The table includes one row per item, and a single column family called **bids**. In each row, there is a column for each bid for that row's item. Column names are composite values representing the bid amount and a bid identifier (a reference to a row in another HBase table). Cells hold composite values identifying the bid quantity and bid date.

The HBase **ItemBids** table can be modeled by the same generic schema that was shown in Figure 4.3. In this case, each row in the HBase table results in *multiple* rows in

```
{ _id:  2315, bids: [
  { _id: b9734, amount: 25,
     quantity: 16, date: "2016-05-02"},
  { _id: b3267, amount: 24,
     quantity: 6, date: "2016-05-02"},
  { _id: b9907, amount: 22,
     quantity: 8, date: "2016-05-01"}
]}
{ _id:  2416, bids: [
  { _id: b7633, amount: 65,
     quantity: 1, date: "2016-04-09"},
  { _id: b9028, amount: 60, date: "2016-04-01"}
]}
```

Figure 4.5: An **ItemBids** collection in MongoDB

the generic relation — one row per bid. To identify the underlying conceptual model, the user must understand that column names are composites of bid values and bid identifiers, and similarly that the cell values are composites. The user must also understand that row keys are item identifiers. This interpretation is commonly imposed on data when it is read from an HBase table by an application. Thus, a user with knowledge of the application can identify attributes either directly from the database or through their application knowledge.

MongoDB: Unlike Cassandra and HBase, MongoDB stores data in collections of documents. Each document in a collection is a JSON object containing at minimum a primary key. The only metadata available from MongoDB is the names of these collections. While each document is permitted to contain arbitrary JSON data, in practice, documents within the same collection have some common structure. Figure 4.5 shows how the same information that is recorded in the HBase table from Figure 4.4 might be represented in a MongoDB document collection, **ItemBids**. Each document contains an ID as well as an array of bids for the item. The ID for each bid references another collection. The **ItemBids** collection could be modeled using the same generic relation schema that was used to model the **ItemBids** table for HBase and Cassandra.

In general, we anticipate that the definition of a generic physical schema for an application will require user involvement. However, there are tools that may assist with this process. For example, several authors have proposed methods for extracting a schema from JSON records in a document store, which could be applied to extract the generic physical schema required for renormalization [66, 75, 125]. These methods generate nested

schemas, but nested properties can be flattened by concatenating their names. Similarly, arrays can be flattened by including multiple rows for each document, as we have done in this example.

4.3 Dependency Input

The second step of the renormalization process is to identify dependencies among the attributes in the generic physical schema. Our normalization algorithm is able to use two types of dependencies: functional dependencies and inclusion dependencies. These two forms of dependencies are easy to express and are commonly used in database design [88].

These dependencies can be input manually, by a user who is familiar with the application and its database. Alternatively, dependencies can be mined from an instance of the underlying NoSQL database. In this section, we describe the types of dependencies that our normalization algorithm expects. We defer discussion of dependency mining to Section 4.5.

Functional dependencies (FDs) are of the form $R : A \rightarrow B$, where R is a relation from the physical schema and A and B are sets of attributes from R. For example, for the **ItemBids** relation described in Section 4.2, the user might identify the following functional dependencies:

$$\begin{aligned} \text{itemID}, \text{bid}, \text{bidID} &\rightarrow \text{quantity}, \text{date} \\ \text{bidID} &\rightarrow \text{itemID}, \text{bid}. \end{aligned}$$

The first may be identified because **itemID**, **bid**, and **bidID** together form a row key for the physical relation. The latter may be identified based on knowledge of the application domain. We expect the functional dependencies provided as input to our algorithm are given in the order they should be processed. That is, the schema will be normalized starting with the first dependency in the list.

Inclusion dependencies (INDs) are of the form $R(A) \subseteq S(B)$ where R and S are physical relations, A is a set of attributes in R and B is a set of attributes in S. The dependency states that for any tuple in R, there exists a tuple in S where the values of attributes in B match the values of the attributes in A for the tuple in R. To represent both the inclusion dependencies $R(A) \subseteq S(B)$ and $S(B) \subseteq R(A)$, we use the shorthand $R(A) = S(B)$. Inclusion dependencies are useful to determine when an application has duplicated attributes across multiple physical structures.

For input to our algorithm, we require that all INDs are superkey-based. That is, for an IND $R(A) \subseteq S(B)$, B must be a superkey of S. We do not believe that this is a significant restriction since we intend for inclusion dependencies to be used to indicate foreign key relationships which exist in the denormalized data. Indeed, Mannila and Räihä [88] have previously argued that only key-based dependencies are relevant to logical design.

4.4 ESON Normalization Algorithm

Levene and Vincent [82] define a normal form for database relations involving functional and inclusion dependencies referred to as inclusion dependency normal form (IDNF). They have shown that normalizing according to IDNF removes redundancy from a database design implied by the set of dependencies. However, one of the necessary conditions for this normal form is that the set of inclusion dependencies is non-circular. A set of inclusion dependencies $\mathrm{I}_1 : R_1(A_1) \subseteq \ldots \subseteq \mathrm{I}_n : R_n(A_n)$ is circular if $R_1 = R_n$. This excludes useful schemas which express constraints such as one-to-one foreign key integrity. For example, for the relations $R(\underline{A}, B)$ and $S(\underline{B}, C)$ we can think of the circular inclusion dependencies $R(A) = S(B)$ as expressing a one-to-one foreign key between $R(A)$ and $S(B)$.

Levene and Vincent also propose an extension to IDNF, termed *interaction-free inclusion dependency normal form* which allows such circularities. The goal of our normalization algorithm is to produce a schema that is in interaction-free IDNF. This normal form avoids redundancy implied by functional and inclusion dependencies while still allowing the expression of useful information such as foreign keys. We provide more details on this normal form in Section 4.4.5. As we show in Section 4.7, relation schemas in this normal form are useful as logical models for several real-world examples.

Figure 4.6 provides an overview of our normalization algorithm, which consists of four stages. In the reminder of this section, we discuss the normalization algorithm in more detail. We will make use of a running example based on the simple generic (denormalized) physical schema and dependencies shown in Figure 4.7.

Our normalization algorithm first applies dependency inference rules, as we discuss in Section 4.4.1. Second, the `BCNFDecompose` algorithm implements BCNF decomposition. This removes any redundancy according to the set of FDs. Next, the `Fold` algorithm removes redundant attributes and relations according to the set of INDs. Finally, `BreakCycles` breaks any inclusion dependency cycles which are not proper circular, to ensure that the resulting schema is in interaction-free IDNF.

Data: A set of relations $\mathbf{R}$, FDs $\mathbf{F}$, and INDs $\mathbf{I}$
Result: A normalized set of relations $\mathbf{R}'''$
begin

```
// Perform dependency inference
F', I+ ← Expand(F, I)
// Normalize according to BCNF
R', I+' ← BCNFDecompose(R,F',I+)
// Remove redundant attributes and relations
R'', I+'' ← Fold(R', F', I+')
// Break remaining circular INDs
R''', I+''' ← BreakCycles(R'', I+'')
```

Figure 4.6: Algorithm for normalization to interaction-free IDNF

4.4.1 Dependency Inference

To minimize the effort required to provide input needed to create a useful normalized schema, we aim to infer dependencies whenever possible. Armstrong [13] provides a well-known set of axioms which can be used to infer FDs from those provided as input. Similarly, Mitchell [99] presents a similar set of inference rules for INDs.

Mitchell further presents a set of inference rules for joint application to a set of FDs and INDs. We adopt Mitchell's pullback and collection rules to infer new functional dependencies for inclusion dependencies and vice versa. As an example of the pullback rule, consider the following dependencies for our example in Figure 4.7:

$$\text{Employees} : \text{EmpID} \rightarrow \text{EmpName}$$
$$\text{EmpProjects}\,(\text{EmpID}, \text{EmpName}) \subseteq \text{Employees}\,(\text{EmpID}, \text{EmpName})\,.$$

In this case, we are able to infer an additional functional dependency:

$$\text{EmpProjects} : \text{EmpID} \rightarrow \text{EmpName}.$$

This case is equivalent to propagating primary keys of different logical entities (in this case, employees) across different relations.

Physical Schema

$$\begin{aligned}\text{EmpProjects}(&\underline{\text{EmpID}}, \text{EmpName}, \underline{\text{ProjID}}, \text{ProjName})\\ \text{Employees}(&\underline{\text{EmpID}}, \text{EmpName}, \text{DeptID}, \text{DeptName})\\ \text{Managers}(&\underline{\text{DeptID}}, \text{EmpID})\end{aligned}$$

Functional Dependencies

$$\begin{aligned}\text{Employees} : \text{EmpID} &\rightarrow \text{EmpName}, \text{DeptID}\\ \text{Employees} : \text{DeptID} &\rightarrow \text{DeptName}\\ \text{EmpProjects} : \text{ProjID} &\rightarrow \text{ProjName}\\ \text{Managers} : \text{DeptID} &\rightarrow \text{EmpID}\end{aligned}$$

Inclusion Dependencies

$$\begin{aligned}\text{EmpProjects}\,(\text{EmpID}, \text{EmpName}) &\subseteq \text{Employees}\,(\ldots)\\ \text{Managers}\,(\text{EmpID}) &\subseteq \text{Employees}\,(\ldots)\\ \text{Employees}\,(\text{DeptID}) &\subseteq \text{Managers}\,(\ldots)\end{aligned}$$

When attributes have the same names, we use ... on the right.

Figure 4.7: Example generic physical schema and dependencies.

The collection rule allows the inference of new inclusion dependencies. Assume the **EmpProjects** relation also contained the **DeptID** attribute. We could then express the following dependencies:

$$\begin{aligned} \text{EmpProjects}\,(\text{EmpID}, \text{DeptID}) &\subseteq \text{Employees}\,(\ldots) \\ \text{EmpProjects}\,(\text{EmpID}, \text{EmpName}) &\subseteq \text{Employees}\,(\ldots) \\ \text{Employees} : \text{EmpID} &\to \text{DeptID} \end{aligned}$$

From this, we could infer the new inclusion dependency

$$\text{EmpProjects}\,(\text{EmpID}, \text{EmpName}, \text{DeptID}) \subseteq \text{Employees}\,(\ldots)\,.$$

This can be seen as collecting all attributes corresponding to a single logical entity. As we will see by example, this allows the elimination of attributes and relations via the `Fold` algorithm to reduce the size of the resulting schema while maintaining the same semantic information.

There is no finite complete axiomatization for FDs and INDs taken together [28, 99]. However, proofs presented in both the cited works rely on the existence of inclusion dependencies which are not superkey-based. These types of dependencies are not permitted by our algorithm. Thus we are uncertain of the completeness of our `Expand` procedure. However, `Expand`, which uses Mitchell's pullback and collection rules for combined inference from FDs and INDs, is sound. It also terminates, since the universe of dependencies is finite and the inference process is purely additive. In practice `Expand` is able to infer dependencies that are useful for schema design. As we noted earlier, INDs which are not key-based are often not considered during schema design.

4.4.2 BCNF Decomposition

The second step, `BCNFDecompose`, is to perform a lossless join BCNF decomposition of the physical schema using the expanded set of FDs. We use a procedure similar to the one described by Garcia-Molina et al. [56].

When relations are decomposed, we project the FDs and INDs from the original relation to each of the relations resulting from decomposition. In addition, we add new inclusion dependencies which represent the correspondence of attributes between the decomposed relations. For example, when performing the decomposition $R\,(ABC) \to R'\,(AB)\,, R''\,(BC)$ we also add the INDs $R'\,(B) \subseteq R''\,(B)$ and $R''\,(B) \subseteq R'\,(B)$. In Appendix B.1, we prove the soundness of these dependency inferences.

Physical Schema

$$\begin{aligned}
\text{Employees}\,(\underline{\text{EmpID}}, \text{EmpName}, \text{DeptID}) \qquad & \text{Departments}\,(\underline{\text{DeptID}}, \text{DeptName}) \\
\text{EmpProjects}\,(\underline{\text{EmpID}}, \underline{\text{ProjID}}) \qquad & \text{EmpProjects}'\,(\underline{\text{EmpID}}, \text{EmpName}) \\
\text{Projects}\,(\underline{\text{ProjID}}, \text{ProjName}) \qquad & \text{Managers}\,(\underline{\text{DeptID}}, \text{EmpID})
\end{aligned}$$

Functional Dependencies

$$\begin{aligned}
\text{Employees} : \text{EmpID} \to \text{EmpName}, \text{DeptID} \qquad & \text{Departments} : \text{DeptID} \to \text{DeptName} \\
\text{Managers} : \text{DeptID} \to \text{EmpID} \qquad & \text{EmpProjects}' : \text{EmpID} \to \text{EmpName} \\
\text{Projects} : \text{ProjID} \to \text{ProjName}
\end{aligned}$$

Inclusion Dependencies

$$\begin{aligned}
\text{EmpProjects}\,(\text{EmpID}) &\subseteq \text{Employees}\,(\ldots) \\
\text{EmpProjects}'\,(\text{EmpID}, \text{EmpName}) &\subseteq \text{Employees}\,(\ldots) \\
\text{EmpProjects}'\,(\text{EmpID}) &= \text{EmpProjects}\,(\ldots) \\
\text{Projects}\,(\text{ProjID}) &= \text{EmpProjects}\,(\ldots) \\
\text{Employees}\,(\text{DeptID}) &\subseteq \text{Departments}\,(\ldots) \\
\text{Managers}\,(\text{DeptID}) &\subseteq \text{Departments}\,(\ldots)
\end{aligned}$$

Figure 4.8: Relations and dependencies after BCNF decomposition.

Note that $=$ is used to represent bidirectional inclusion dependencies.

In our running example, we are left with the relations and dependencies shown in Figure 4.8 after the `Expand` and `BCNFDecompose` steps. The **Employees** relation has been decomposed to add **Departments**. Also, the **EmpProjects** relation has been decomposed to add **EmpProjects′** and **Projects**. For illustrative purposes, we have manually given new relations sensible names. In practice, the user would need to choose relation names once the normalization process is complete.

4.4.3 Folding

Casanova and de Sa term the technique of removing redundant relations *folding* in the context of conceptual schema design [27]. Our algorithm, `Fold` (Figure 4.9), identifies any

```
Function Fold(R, I) is
    Data: A set of relations R, FDs F, and INDs I
    Result: A new set R′ without redundant attributes/relations
    R′ ← R
    do
        // Remove redundant attributes
        foreach IND R(A) ⊆ S(B) in I do
            // Find FDs implying attributes in R are redundant
            foreach FD C → D | CD ⊆ A in F do
                // Remove attributes which are in the RHS of the FD
                R′ ← R′ \ {R} ⋃ {R(A \ D)}
        // Remove redundant relations
        foreach IND pair R(A) = S(B) in I do
            if R(A) = R then
                // All attributes are also in the other relation
                R′ ← R′ \ R
            if Pk(R(A)) = Pk(S(B)) then
                // Merge R and S
                T ← A ⋃ B
                if R′ \ {R, S} ⋃ {T} is in BCNF then
                    R′ ← R′ \ {R, S} ⋃ {T}
    until R′ is unchanged from the previous iteration;
```

Figure 4.9: Relation folding based on INDs

attributes or relations which are recoverable from other relations, based on the INDs. These attributes and relations are redundant and the `Fold` algorithm removes them from the schema. More abstractly, folding removes attributes which can be recovered by joining with another relation and relations which are redundant because they are simply a projection of other relations. `Fold` also identifies opportunities for merging relations that share a common key.

It is not necessary to perform the `Fold` step to ensure that the resulting schema is in interaction-free IDNF. However, if two schemas contain equivalent information, we believe the smaller is more useful as it is a more concise representation of the application domain and does not result in any loss of information. We do not make any claims that `Fold` produces a *minimal* schema in interaction-free IDNF, but the opportunities for reduction we have identified are useful in practice. For example, consider the **EmpProjects′** relation

which contains the **EmpName** attribute. Since we have the inclusion dependency

$$\text{EmpProjects}'\,(\text{EmpID}, \text{EmpName}) \subseteq \text{Employees}\,(\ldots)$$

and the functional dependency

$$\text{Employees} : \text{EmpID} \to \text{EmpName}$$

we can infer that the **EmpName** attribute in **EmpProjects′** is redundant since it can be recovered by joining with the **Employees** relation. With the **EmpName** attribute removed, we see that we have the inclusion dependency

$$\text{EmpProjects}'\,(\text{EmpID}) = \text{EmpProjects}\,(\ldots)\,.$$

Since **EmpProjects′** is simply a projection of an attribute from **EmpProjects**, the **EmpProjects′** relation can be removed. It is important to note that the inclusion dependencies are bidirectional so that the exact set of tuples represented by the relation being removed is recoverable.

Finally, we consider an example of merging. Suppose the original schema contained another relation storing the addresses of all employees, $\text{EmpAddress}\,(\underline{\text{EmpID}}, \text{Address})$. Assuming we had an address for each employee, we can express the inclusion dependency

$$\text{Employees}\,(\text{EmpID}) = \text{EmpAddress}\,(\ldots)\,.$$

We can then merge **Employees** and **EmpAddress** by adding the **Address** attribute to the **Employees** relation since they share a common key.

Lemma 1. *`Fold` does not introduce any BCNF violations.*

Proof. When removing relations with `Fold`, clearly no BCNF violations are created. The attributes removed by `Fold` are never keys of the relation, so they also do not introduce BCNF violations. When attempting to merge relations via `Fold` we explicitly avoid merging if a BCNF violation would be introduced. □

4.4.4 Breaking IND Cycles

Mannila and Räihä [88] use a technique, which we call `BreakCycles` (Figure 4.10), to break circular inclusion dependencies when performing logical database design. We adopt this technique to break inclusion dependency cycles which are not proper circular.

Function `BreakCycles(`$\boldsymbol{R}$, $\boldsymbol{I}$`)` **is**

Data: A set of relations $\mathbf{R}$ and INDs $\mathbf{I}$

Result: A set of relations $\mathbf{R}'$ with cycles removed and new dependencies $\mathbf{I}^+$

$\mathbf{R}' \leftarrow \mathbf{R}$

foreach *Set of circular INDs* $R_1(X_1) \subseteq R_2(Y_2) \cdots \subseteq R_n(X_n) \subseteq R_1(Y_1)$ *in* $\boldsymbol{I}$ **do**

$R_1' \leftarrow X_1 Y_1$

$R_1'' \leftarrow Y_1 + \text{attr}(R_1) - X_1 Y_1$

$\mathbf{R}' \leftarrow \mathbf{R}' \setminus \{R_1\} \bigcup \{R_1', R_1''\}$

$I^+ \leftarrow I^+ \bigcup R_1'(X_1) \subseteq R_2(Y_2)$

$I^+ \leftarrow I^+ \bigcup R_1'(Y_1) \subseteq R_1''(Y_1)$

$I^+ \leftarrow I^+ \bigcup R_n(X_n) \subseteq R_1''(Y_1)$

$I^+ \leftarrow I^+ \setminus \{R_1(X_1) \subseteq R_2(Y_2), \subseteq R_n(X_n) \subseteq R_1(Y_1)\}$

Figure 4.10: Breaking circular inclusion dependencies

In our running example, we have an inclusion dependency cycle which is not proper circular created by the following two INDs:

$$\text{Managers}(\text{EmpID}) \subseteq \text{Employees}(\ldots)$$
$$\text{Employees}(\text{DeptID}) \subseteq \text{Managers}(\ldots).$$

Applying the `BreakCycles` algorithm removes **DeptID** from the **Employees** relation and adds a new relation WorksIn(EmpID, DeptID). We then add the following inclusion dependencies to the **WorksIn** relation:

$$\text{WorksIn}(\text{EmpID}) \subseteq \text{Employees}(\ldots)$$
$$\text{WorksIn}(\text{DeptID}) \subseteq \text{Managers}(\ldots).$$

The inclusion dependency Employees(DeptID) $\subseteq$ Managers(...) is also removed, breaking the cycle.

Lemma 2. *`BreakCycles` does not introduce any BCNF violations.*

Proof. `BreakCycles` decomposes a relation into two relations with the only functional dependency defined establishing a primary key. The only new dependencies added are inclusion dependencies between corresponding attributes in the decomposed relations. This does not permit the inference of any new functional dependencies. Therefore, the final schema is still in BCNF with respect to $\mathbf{F}'$. □

4.4.5 IDNF

The goal of our normalization algorithm is to produce a schema that is in interaction-free IDNF with respect to the given dependencies. The following conditions are given by Levene and Vincent [82] as the definition of interaction-free IDNF with respect to relations **R** and a set of FDs **F** and INDs **I**:

1. **R** is in BCNF [42] with respect to **F**.
2. All the INDs in **I** are key-based or proper circular.
3. **F** and **I** do not interact (the notion of interaction is explained further below).

A set of INDs is proper circular if for each circular inclusion dependency over a unique set of relations $R_1(X_1) \subseteq R2(Y_2), R_2(X_2) \subseteq R_3(Y_3), \ldots, R_m(X_m) \subseteq R_1(Y_1)$, we have $X_i = Y_i$ for all i.

Lemma 3. *The schema produced by the normalization algorithm of Figure 4.6 is in interaction-free IDNF with respect to the given sets of FDs and INDs.*

Proof. Rule 1 is satisfied because `BCNFDecompose` produces a schema that is BCNF with respect to $\mathbf{F}'$, and therefore with respect to **F**. Furthermore, the subsequent `Fold` and `BreakCycles` algorithms do not introduce any BCNF violations.

Rule 2 states that all remaining INDs must be key-based. The given set of INDs (**I**) is superkey-based by assumption. We can show that all of the additional INDs created by the algorithm are also key-based. Furthermore, none of the schema transformations can result in non-key-based INDs. A complete proof of this is given in Appendix B.3.

To show that the final schema satisfies rule 3 (non-interaction of FDs and INDs), we make use of a sufficient condition for non-interaction given by Levene and Vincent [82]. A set of FDs **F** and INDs **I** over a set of relations do not interact if the relations are in BCNF with respect to **F**, **I** is proper circular, and $\mathbf{F} \bigcup \mathbf{I}$ is *reduced*. As stated above, the final schema is in BCNF. All inclusion dependencies are proper circular since we explicitly break any cycles which are not via the `BreakCycles` algorithm. It remains to show that the set of functional and inclusion dependencies are reduced.

A set of functional inclusion dependencies **F** and **I** is reduced if for every inclusion dependency $R(X) \subseteq S(Y)$, there are only trivial functional dependencies involving attributes in the set Y. We have already shown that the final set of inclusion dependencies is key-based, implying that Y is a key of S. Since the set Y is a key, **F** can only contain trivial functional dependencies involving Y. Therefore, $\mathbf{F} \bigcup \mathbf{I}$ is reduced and the schema is in interaction-free IDNF. □

4.5 Dependency Mining

As we noted in Section 4.3, the dependencies required by our algorithm can be directly specified by a knowledgeable user, or mined from an instance of the underlying database. In this section, we consider the latter option in more depth.

4.5.1 Mining for NoSQL Normalization

Dependency mining tools operate by examining a database instance to discover all dependencies that hold on it. In order to mine dependencies, such tools collect statistics on the distribution of values in each column in an attempt to discover relationships. We make use of Apache Calcite [4] to provide an interface between NoSQL databases and dependency mining tools, so that the tools can obtain the metadata and statistics they require. Calcite is a data management framework which presents a SQL query interface on top of a variety of database backends. We have used TANE [65] for mining functional dependencies and BINDER [105] for mining inclusion dependencies. These algorithms both produce all valid dependencies which hold on the given instance.

The problem with using mined dependencies is that many of them will be spurious. For example, suppose in the employees relation of a company database we have the functional dependency $\text{Department}, \text{Salary} \rightarrow \text{FirstName}$. This expresses that all employees in a department with the same salary have the same first name. While this might hold for some particular instance of the schema, it is unlikely to represent semantically meaningful information. Another valid dependency on the same relation may be $\text{DeptID} \rightarrow \text{Department}$. We would prefer to perform BCNF decomposition using the second dependency since it would result a table with a primary key of DeptID, which is likely to be more useful than one with key $(\text{Department}, \text{Salary})$.

We use two techniques to reduce the impact of spurious dependencies: 1) ranking of functional dependencies for the selection of primary keys for a relation and 2) ranking of functional dependencies for the selection of a BCNF-violating dependency for decomposition. Any time we generate a new relation, we use heuristics to select a primary key. Functional dependencies representing other possible candidate keys are then ignored when performing BCNF decomposition. Instead, we use heuristics to rank the remaining dependencies to select the next violating functional dependency to use for decomposition. Note that we still consider all valid functional dependencies which violate BCNF. However, by selecting a good order for decomposition, some spurious functional dependencies no longer

apply when their attributes are split into separate relations. Avoiding decomposition based on these dependencies results in a more logical schema as output.

We make use of three heuristics to identify functional dependencies which are likely to represent keys:

1. **Length**: Keys with fewer attributes
2. **Value**: Keys with shorter values
3. **Position**: Keys occurring further left in the relation definition without non-key attributes between key attributes.

We use these heuristics for both primary key and violating dependency selection. Since we target NoSQL databases, we do not blindly apply the value length heuristic to all columns. This is because data types exist which are explicitly intended to represent unique identifiers. For example, Cassandra allows columns of type UUID and MongoDB documents can have values of type ObjectId. These are both long pseudorandom values intended to allow concurrent creation without collision. Thus, although the values are long, we know that these values are likely to represent key attributes. We assign such columns the highest score according to the Value heuristic.

Papenbrock and Naumann [106] used similar heuristics in an algorithm for BCNF normalization of a schema using mined functional dependencies. (They did not consider identifier types, since they did not target NoSQL databases.) They also propose an additional heuristic which measures duplication across sets of column values in a dependency. We did not use this heuristic since it increases complexity by requiring joint statistics across multiple columns, and our algorithm produces positive results without this heuristic.

4.6 Applications of the Logical Model

The logical schema produced by the renormalization process is useful as a form of documentation of the information that is embodied, in denormalized form, in a NoSQL database. However, the logical schema has other applications as well. Our original motivation for this work was to be able to provide a conceptual model of an existing NoSQL database as input to a NoSQL schema design tool, such as NoSE. Given a conceptual model of the database, as well as a description of the application workload, NoSE generates a physical schema

Logical schema query

```
SELECT EmpName, ProjID, ProjName FROM Projects
NATURAL JOIN Employees WHERE EmpName = ?
```

Physical schema query

```
SELECT EmpName, ProjID, ProjName FROM EmpProjects WHERE EmpName = ?
```

Figure 4.11: Query rewriting against the logical schema

optimized to support that workload. By combining renormalization with a schema design tool, we can optimize the physical schema design of an existing NoSQL-based application.

It may also be useful to express application queries and updates directly against the logical model. This can provide a means of executing new, ad-hoc queries over an existing NoSQL database without the need to understand how the data is denormalized. In the remainder of this section, we discuss how we can execute queries expressed over the logical model using information gathered during the execution of our normalization algorithm.

4.6.1 Ad-Hoc Query Execution

One of the main advantages of using dependency information to construct the logical schema is that we can use the same information to assist with executing queries written against the logical schema. Because NoSQL databases often lack the ability to perform any complex processing of queries, developers express queries directly in terms of structures from the physical schema. This tightly couples the application to a particular schema and makes changes in the schema difficult. With an appropriate logical schema for the application, we can rewrite queries written against this logical schema to target specific physical structures as in Figure 4.11. In this case, we can identify that the **EmpProjects** relation materializes the join in the logical schema query and is therefore able to provide an answer. Our aim is for this rewriting to happen transparently and to enable the possibility of changing the rewriting as the physical schema changes.

As we show in the following section, we can produce queries on the logical schema which correspond to data stored in the original structures in the physical schema. We can think of these queries as defining materialized views over the logical schema which correspond to the physical schema. The application developer can use these queries directly in cases where the application directly used data from these structures without additional manipulation. This simplifies rewriting existing application queries if a developer wishes to move to using the logical model. For more complex queries, we can use existing techniques to rewrite the

queries to make use of the materialized views [60].

These queries can be translated on-the-fly to enable ad-hoc query execution. For example, Apache Calcite [4] is a dynamic data management framework which connects to different backends, including those for NoSQL datastores. We are currently exploring rules for view-based query rewriting in Calcite to enable the necessary transformations. We leave a full implementation of this approach as future work.

4.6.2 View Definition Recovery

In order to allow logical queries to execute against the existing physical schema, we must have a way of understanding how the existing physical structures map to the logical schema. Fortunately, we can use information saved from the normalization process to produce this mapping. We simply think of each physical structure in the original schema as a materialized view. We can recover a query which serves as the materialized view definition by tracking a small amount of additional information during the normalization process.

For an example of view definition recovery, consider the **EmpProjects** relation from Figure 4.7. Before performing normalization, our set of relations is equivalent to the input so our view definition for **EmpProjects** is `SELECT EmpID, EmpName, ProjID, ProjName FROM EmpProjects`. Considering only the **EmpProjects** relation, we have the following functional dependencies:

$$\text{EmpID} \rightarrow \text{EmpName and}$$
$$\text{ProjID} \rightarrow \text{ProjName.}$$

When we perform BCNF decomposition, the normalization algorithm splits **EmpProjects** into three relations. We call the relation with employee data **EmpProjects′**, the relation with project data **Projects**, and keep the remaining relation expressing the association with the name **EmpProjects**. We can now write the view definition to include a join based on the decomposition. Our view definition then appears as below:

```
SELECT EmpID, EmpName, ProjID, ProjName FROM EmpProjects JOIN EmpProjects'
  ON EmpProjects.EmpID = EmpProjects'.EmpID
  JOIN Projects ON EmpProjects.ProjID = Projects.ProjID.
```

A similar process of creating joins applies when running the `BreakCycles` algorithm. The other transformation which affects the view definitions is `Fold`. When removing

relations, the transformation is a simple rename of the relation in the view definition. For example, after the `Fold` step of our algorithm is performed on the **EmpProjects′** relation, we see that it can be removed as was discussed in Section 4.4.3. This is because the data in **EmpProjects′** can be recovered from the **Employees** relation. We can simply replace all instances of **EmpProjects′** in the definition above with **Employees**. We do not show an example, but a similar renaming applies when `Fold` removes an attribute with the addition that a join is also created involving the relation which contains the removed attribute.

For a relation R, we can recover the list of logical structures it references by recursively visiting the list of relations decomposed to produce R until we reach physical structures from the original schema. Since all of our inclusion dependencies are superkey-based, all of the view definitions will consist of foreign key joins. More specifically, a materialized view definition for the relation R will be of the form `SELECT` $attr\,(R)$ `FROM` R_1 `JOIN` R_2 `ON` $R_1.A = R_2.B \cdots$ `JOIN` $R_{n-1}.X = R_n.Y$ where $attr\,(R)$ is a list of the attributes in R and R_1 through R_n are the relations the query must join.

Using these materialized view definitions, we can answer queries written against the logical schema using view-based query rewriting as discussed in the previous section. The goal of view-based query rewriting is to answer a query using a set of materialized views, which is exactly what we are trying to accomplish. We note that the view definitions we described above are all conjunctive queries. It has recently been shown that conjunctive query determinacy is undecidable in general [59]. However, there are useful subclasses of conjunctive queries for which determinacy is decidable [107]. This suggests the possibility that these rewriting techniques may be useful for answering queries written against logical schemas produced by ESON.

4.7 Case Studies

This section presents case studies of several denormalized database schemas to show how ESON is able to recover a useful schema. We discuss both cases where dependencies were specified manually and where the dependencies were mined using an instance of the denormalized schema.

4.7.1 RUBiS

RUBiS [30], a Web application for online auctions was introduced in Chapter 3 We presented a tool called NoSE, which performs automated schema design for NoSQL systems. We used NoSE to generate two Cassandra schemas for RUBiS, each optimized for a different workload (a full description is given in Appendix C). In each case, NoSE starts with a conceptual model of the RUBiS database, The conceptual model includes six types of entities (e.g., users, and items) with a variety of relationships between them. The first physical design consists of 9 Cassandra column families, while the second, larger design has 14 column families.

As our first case study, we used NoSE's denormalized Cassandra schemas as input to our normalization algorithm so that we can compare the normalized schemas that it produces with the original conceptual schema that NoSE started with. For each physical schema, we tested our algorithm with two different sets of dependencies: one set manually generated from the physical schema, and a second set mined from an instance of that schema using the mining technique discussed in Section 4.5. This resulted in a total of four tests.

For both schemas, renormalization using manually identified dependencies resulted in a conceptual model that was identical (aside from names of relations and attributes) to the original conceptual schema used by NoSE, as desired.

For the two tests with mined dependencies, the renormalization program produced the original conceptual schema, as desired, in the case of the smaller (9 column family) Cassandra schema, but not in the case of the larger (14 column family) Cassandra schema. For the smaller schema, the mining process identified 61 functional dependencies and 314 inclusion dependencies. The dependency ranking heuristics were critical to this success. Without them, spurious dependencies lead to undesirable entities in the output schema. For example, one contains only the fields **BidValue** and **BidQuantity**, which is not a semantically meaningful entity. For the larger schema, mining found 86 functional dependencies and 600 inclusion dependencies, many of them spurious. In this case, the ranking heuristics were not sufficient to eliminate undesirable decompositions during renormalization. No set of ranking heuristics will be successful in all cases, but it is clear that this is an important area for improvement in future work.

The large schema test with manually chosen dependencies provided a good example of relation merging using `Fold` step of our algorithm. In the conceptual schema, there is a **Comments** entity set which has relationships to the user sending and receiving the comment. The denormalized schema has two separate relations which store the comments according to the sending and receiving users.

After performing BCNF decomposition, we end up with relations similar to the following (simplified for presentation):

$$\text{CommentsSent}\,(\text{id}, \text{sending\_user}, \text{text})$$
$$\text{CommentsReceived}\,(\text{id}, \text{receiving\_user})\,.$$

We also have inclusion dependencies which specify that the `id` attribute in both relations is equivalent, i.e. $\text{CommentsSent}\,(\text{id}) = \text{CommentsReceived}\,(\ldots)$. Since the key of these relations is equivalent, the `Fold` algorithm will merge these two relations producing Comments(id, receiving_user, sending_user, text).

These examples show that functional and inclusion dependencies are able to drive meaningful denormalization. Runtime for the normalization step of our algorithm was less than one second on a modest desktop workstation in all cases.

4.7.2 MongoDB

Stolfo [116] presents a case study of schema design in MongoDB to explore design alternatives. We extract a design from the examples to show how our normalization process can produce a suitable logical model. The system being designed is for library management and deals with patrons, books, authors, and publishers. While the case study shows different possible schemas, we have selected one for demonstration purposes and we present example documents for this schema below. This model contains a significant amount of denormalized data inside the collection of patron documents.

As described in Section 4.2, we first manually defined a physical relational schema capturing the information in the MongoDB database documents. This is shown in Figure 4.12. The MongoDB patron documents included an array of book loans for each patron. To produce the physical schema, we flattened the loan attributes into the **Patron** relation, and added the key of each array element as part of the superkey the **Patron** relation. We also manually identified a (non-exhaustive) set of functional and inclusion dependencies over these relations, as shown in Figure 4.13.

The denormalization in this schema consists of the duplication of patron, book, and author information in the patron documents. For this application, the algorithm was able to identify the denormalization and produce a logical model without duplication. The logical schema produced by our normalization algorithm removes this denormalization. We note that the dependencies in Figure 4.13 do not contain any functional dependencies involving loans, which were nested in patron documents in MongoDB database. However,

Publishers(<u>_id</u>, name, founded, <u>book</u>)
Books(<u>_id</u>, title, <u>author</u>)
Patrons(<u>_id</u>, name, address.city, address.state,
<u>loans._id</u>, loans.title,
<u>loans.author._id</u>, loans.author.name)
Authors(<u>_id</u>, name)

Figure 4.12: Physical relations from MongoDB schema

$$
\begin{aligned}
\text{Publishers} &: \text{\_id} \rightarrow \text{name, founded, book} \\
\text{Books} &: \text{\_id} \rightarrow \text{title, author} \\
\text{Patrons} &: \text{\_id} \rightarrow \text{name, address.city,} \\
&\qquad \text{address.state, loans.\_id} \\
\text{Authors} &: \text{\_id} \rightarrow \text{name}
\end{aligned}
$$

$$
\begin{aligned}
\text{Publishers(book)} &\subseteq \text{Books (\_id)} \\
\text{Books(author)} &\subseteq \text{Authors (\_id)} \\
\text{Patrons(loans.\{\_id, title,} & \\
\text{author.\_id\})} &\subseteq \text{Books (\_id, title, author)} \\
\text{Patrons(loans.author.} & \\
\text{\{\_id, name\})} &\subseteq \text{Authors (\_id, name)}
\end{aligned}
$$

Figure 4.13: Dependencies on MongoDB physical relations

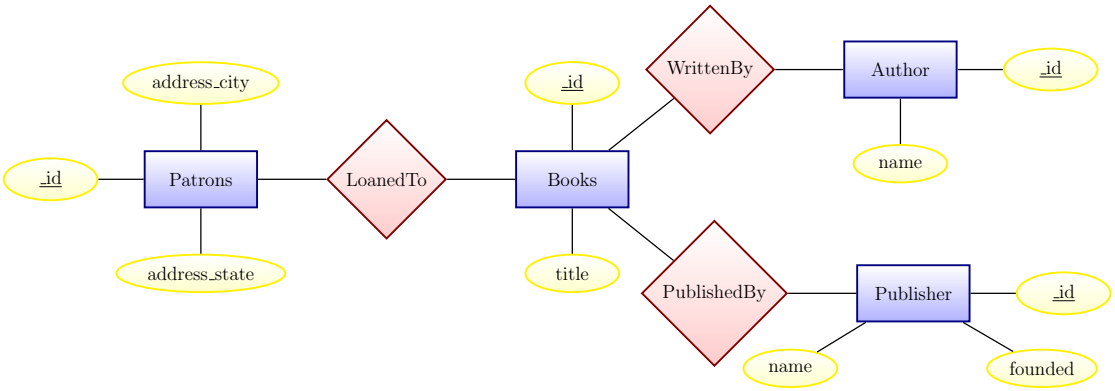


Figure 4.14: MongoDB example schema entities

the `Expand` step of our algorithm is able to infer such functional dependencies based on the inclusion dependencies between **Patrons** and **Authors**/**Books** and the FDs on those relations. The `BCNFDecompose` step separates the redundant title and author information from the **Patrons** relation using these FDs. Finally, the `Fold` step removes this data since it is duplicated in the **Authors** and **Books** relations. This removes all redundancy which was present in the original schema in Figure 4.12. Relationships between publishers, authors and their books were also recovered. The final schema represented as an ER diagram is shown in Figure 4.14.

4.7.3 Twissandra

Twissandra [54] is a simple clone of the Twitter microblogging platform using Cassandra as a database backend. The application stores data on only two different entities: users and tweets. Each tweet has an associated user who created the tweet and each user can "follow" any number of other users.

The Twissandra schema consists of six column families. There is one for both users and tweets keyed by their respective IDs. Two additional column families store the users a particular user is following and separately, their followers. Denormalizing this relationship allows efficient retrieval of users in both directions. Finally, there is a column family storing all data on tweets by user and a column family which stores tweets for all users a user is following. Both of these final two column families contain denormalized data on tweets. The relations corresponding to these column families are given in Figure 4.15. Note that some of the keys identified are in fact superkeys of the associated relation.

users (uname, password)
following (uname, followed)
followers (uname, following)
tweets (tweet_id, uname, body)
userline (tweet_id, uname, body)
timeline (uname, tweet_id, posted_by, body)

Figure 4.15: Twissandra physical relations

Figure 4.16 shows all the functional and inclusion dependencies which we can express over the Twissandra schema. Note that we use $=$ to denote a bidirectional inclusion dependency and we omit attribute names from the right-hand side of inclusion dependencies when the attribute names are the same as on the left-hand side.

Figure 4.17 shows a complete ER diagram for Twissandra, which represents the desired result. However, the conceptual model produced by our normalization process includes one additional relation aside from users and tweets. The conceptual schema produced by our algorithm is still fully normalized in interaction-free IDNF. The additional relation occurs because the normalization process is unable to remove the timeline column family, despite the fact that the information contained in this column family is still redundant. We can reconstruct the timeline relation with the query

```
SELECT f.uname, t.tweet_id, t.uname AS posted_by, t.body
FROM following f JOIN tweets t ON t.followed = t.uname.
```

This redundancy remains because the dependency defining the timeline table cannot be expressed using functional or inclusion dependencies. Expressing this denormalization requires a dependency language which can reference more than two relations. In this case, the dependency is between timeline, followers, and tweets. Extending our dependency language and normalization process to include additional dependencies such as join dependencies [52] would enable us to resolve such issues.

$$
\begin{aligned}
\text{tweets} : \text{tweet\_id} &\to \text{uname}, \text{body} \\
\text{userline} : \text{tweet\_id} &\to \text{uname}, \text{body} \\
\text{timeline} : \text{tweet\_id} &\to \text{posted\_by}, \text{body} \\
\text{users} : \text{uname} &\to \text{password}
\end{aligned}
$$

$$
\begin{aligned}
\text{followers}\,(\text{uname}, \text{following}) &= \text{following}\,(\text{followed}, \text{uname}) \\
\text{userline}\,(\text{uname}) &\subseteq \text{users}\,(\ldots) \\
\text{timeline}\,(\text{uname}) &\subseteq \text{users}\,(\ldots) \\
\text{timeline}\,(\text{posted\_by}) &\subseteq \text{users}\,(\text{uname}) \\
\text{timeline}\,(\text{uname}, \text{posted\_by}) &\subseteq \text{following}\,(\ldots, \text{followed}) \\
\text{timeline}\,(\text{posted\_by}, \text{uname}) &\subseteq \text{followers}\,(\text{following}, \ldots) \\
\text{userline}\,(\text{tweet\_id}, \text{uname}, \text{body}) &= \text{tweets}\,(\ldots)
\end{aligned}
$$

Figure 4.16: Dependencies on Twissandra physical relations

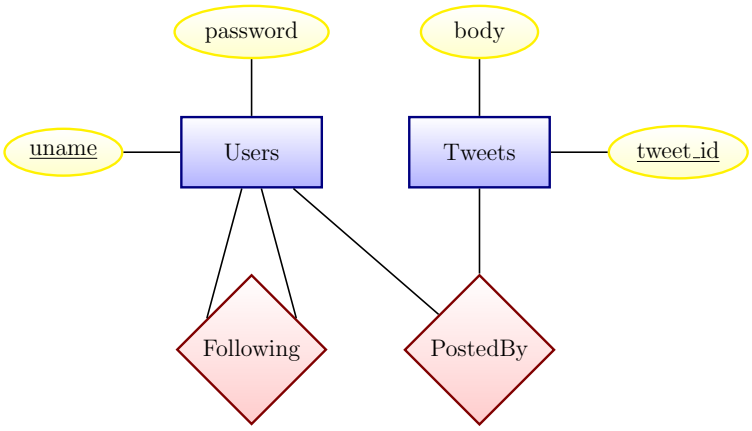


Figure 4.17: Twissandra schema entities

4.8 Summary of ESON

We have developed a methodology for transforming a denormalized physical schema in a NoSQL datastore into a normalized logical schema. Our method makes use of functional and inclusion dependencies to remove redundancies commonly found in NoSQL database designs. We further showed how we can make use of dependencies which were mined from a database instance to reduce the input required from users. Our method has a variety of applications, such as enabling query execution against the logical schema and guiding schema evolution and database redesign as application requirements change.

There are additional opportunities for further automation of NoSQL schema management tasks. One limitation of the logical schema produced by our algorithm is that the relations are not necessarily given meaningful names. Some heuristics such as looking for common prefixes in attribute names may be useful. Currently we also require developers to modify applications manually after we produce the logical model. However, previous work such as Query By Synthesis [35] has shown that it is possible to extract higher-level query patterns from imperative application code. A similar approach could be applied to extract queries from applications which could then be rewritten to use the logical model. We also discussed the possibility of a query execution engine which could transparently retarget these queries to operate on different physical models. We expect a combination of these techniques to improve schema management for NoSQL databases.

Chapter 5

Cache Design for Data Processing Systems

Chapters 3 and 4 dealt with problems surrounding physical design in NoSQL databases. We now explore similar problems in the context of another class of non-relational data systems: distributed data processing frameworks. Users of many of these systems face similar challenges to those faced by users of NoSQL databases. Lack of mature design processes and tools require users to become familiar with the details of individual systems in order to extract high performance. Specifically, we explore physical design techniques that enable non-expert users to write high performance iterative applications for Apache Spark. The problems discussed in Chapters 3 and 4 are not directly applicable to Spark since it does not directly store data. Instead we consider the problem of caching intermediate results to minimize execution time.

5.1 Motivation

Apache Spark [129] is a data processing framework that programmatically constructs a graph of transformations on input data. These transformations are *lazy* in the sense that they are only evaluated when output must be produced. Each evaluation results in a single Spark job which executes all the necessary transformations to produce the output. This allows Spark to pipeline transformations and significantly increase overall efficiency. Each dataset in a Spark pipeline is referred to as a resilient distributed dataset (RDD). RDDs are partitioned to allow for distributed execution across multiple executor nodes. Spark

programs may keep references to any RDD and decide to reuse the same RDD multiple times. To speed up such use cases, Spark allows programs to specify that an RDD should be cached upon first use by annotating it with a `persist` directive (Spark uses persist as a synonym for cache). Furthermore, when a program decides it is no longer going to use an RDD it can specify that the RDD can be removed from the cache, with a separate `unpersist` annotation.

Caching is important for performance, especially for applications making use of iteration, which is a common feature of Spark programs. Unfortunately, caching annotations are difficult to apply correctly, as we show in Section 5.2. Importantly, Spark has no understanding of high-level patterns such as iteration. This requires developers to pay attention to patterns of reuse across iterations and make appropriate use of caching. Spark's lack of knowledge removes opportunities for optimization and results in additional programmer effort required in order to maintain high performance for iterative Spark jobs. We propose an approach to expose the iterative nature of computations to the Spark runtime in order to enable Spark to optimize use of the cache for common cases without developer intervention.

Our work makes the following contributions:

- In Section 5.3.1, we introduce *explicit iteration*, a method of explicitly representing iterative applications in Spark to automate the selection of intermediate results to persist.
- Section 5.3.2 presents *lazy unpersist*, an alternative to the default mechanism of unpersisting cache entries in Spark which ensures that cached results are fully utilized before they are removed from the cache.
- Finally, in Section 5.4 we analyze the performance of our techniques and demonstrate that explicit persist/unpersist annotations by Spark application developers are not always required to achieve good performance.

5.2 The Problem

In the following section, we discuss the challenges faced by developers making use of Spark for iterative computation. Section 5.2.1 describes the types of applications we are aiming to optimize and the problems developers face with their implementation. In Section 5.2.2 we show how this problem is partially addressed via caching of intermediate results, but we also show how simple approaches to caching fall short.

5.2.1 Iterative Computation in Spark

A simple example of a Spark application using iteration is given in Figure 5.1a. The program consists of a series of `map` transformations followed by a `count` action (which forces evaluation). Since Spark uses lazy evaluation and there are no actions within the loop, Spark simply constructs a directed acyclic graph (DAG) of these transformations while the loop runs. That is, Spark simply records metadata about how to compute the RDD for one iteration from the previous iteration. This metadata defines the *lineage* of the RDD. An action such as the `count` on the last line of the program schedules a job which immediately evaluates and executes all of the pending transformations. Since the example program consists of a single action and no aggregation, all transformations are pipelined together. This results in consistently fast execution as shown in Figure 5.2. All benchmarks in this figure were executed on a data set of 100 million randomly generated points using the same hardware described in Section 5.4.[1]

Unfortunately, small changes to the application code can result in a significant degradation of application performance. Consider Figure 5.1b, which shows a variant of the sample program from Figure 5.1a. In this version, the developer wants a data-defined termination condition: stopping when the size of the RDD drops below a specified value. An action (`count`) is now executed for each iteration. Because of this, execution of the application will result in one Spark job per iteration, rather than a single job as was the case for variant V1. Computation of the current value of the `data` RDD will be scheduled on each iteration of the loop. In addition to the transformations no longer being lazy, Spark will also reevaluate all the transformations for the previous iteration. In general, iteration i also evaluates the transformations for iteration $i-1$, which requires evaluating iteration $i-2$, and so on.

The seemingly minor change from variant V1 to variant V2 leads to substantial change in execution time as shown in Figure 5.2. The result is quadratic runtime instead of the linear runtime that would be achieved without the intervening action. This can be resolved through the use of Spark's caching mechanisms, but this requires explicit programmer annotation. In the following section, we show how performance issues can be resolved via caching, which currently requires explicit programmer annotation in Spark.

[1] Spark was configured with 4 executors each with 16GB of memory and an additional 32GB of memory allocated to the driver program.

```
1 // data refers to a Spark RDD
2 for (i <- 1 to iterations) {
3   data = data.map(...).filter(...)
4 }
5 data.count()
```

(a) Variant V1

```
1 // data refers to a Spark RDD
2 for (i <- 1 to iterations) {
3   data = data.map(...).filter(...)
4   if (data.count() < limit) break
5 }
6 data.count()
```

(b) Variant V2: With data-dependent termination

```
1 // data refers to a Spark RDD
2 for (i <- 1 to iterations) {
3   val oldData = data
4   data = data.map(...).filter(...)
5   data.persist()
6   if (data.count() < limit) break
7   oldData.unpersist()
8 }
9 data.count()
```

(c) Variant V3: With data-dependent termination and caching

Figure 5.1: Three variants of a sample iterative Spark application

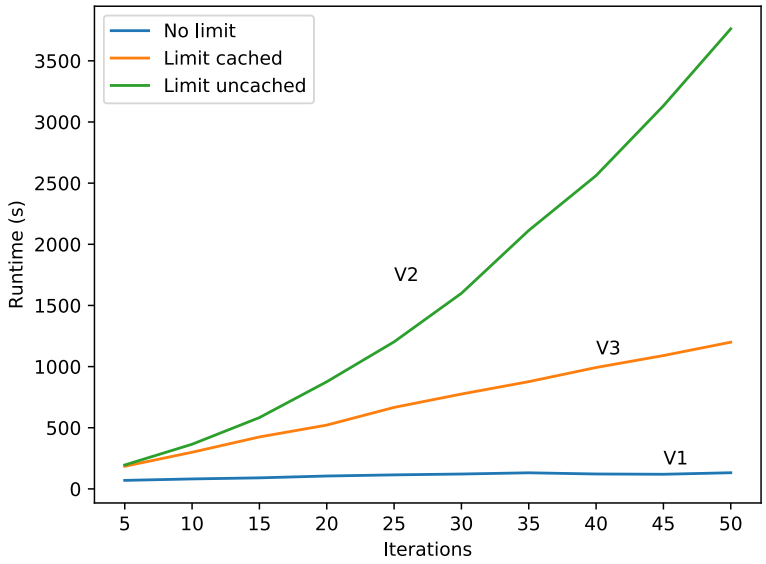


Figure 5.2: Runtime for the Spark application from Figure 5.1

5.2.2 Caching

Spark programmers are expected to explicitly indicate when computed RDDs should be persisted. When writing a Spark application, a developer can store a reference to any RDD in a variable in the host programming language, such as the variable `data` in our sample program. This enables datasets constructed from the same set of transformations to be used multiple times with other additional transformations appended. By default, Spark does not attempt to reuse computed RDDs even when their lineage is identical. A persistence annotation in Spark specifies that the first time an RDD is computed, results of the calculation should be stored for later reuse. When the RDD is first evaluated, it is placed into an internal cache in either memory or disk as specified by the programmer. If the same RDD is reused in a different job, the executor running the job will check its local cache. If the RDD was previously computed and has not been evicted, the partition associated with the RDD will be read from the cache instead of being recomputed.

Thus, to improve variant V2, a Spark developer could choose to persist the results of each iteration as in Figure 5.1c. With this explicit call to `persist`, results of the previous iteration will be stored in Spark's in-memory cache. Therefore, when the next iteration is computed, this data can be retrieved from the cache instead of recomputing

it. Information which is no longer required can be removed from the cache via an explicit call to `unpersist`. The addition of these persistence annotations returns the application to linear runtime (with some additional overhead required for the action to be computed in each iteration compared to variant V1). The effect of enabling caching in the sample program is shown in Figure 5.2.

Although `persist` and `unpersist` annotations are important to the performance of iterative applications, they can be difficult for programmers to apply correctly. A developer must understand the interaction between lazy evaluation, caching, and job scheduling in order to make use of the cache effectively. These interactions can be complex and many Spark users encounter performance bugs as a result of suboptimal use of caching. In particular, the simple approach of caching before RDDs are reused and removing them from the cache (via `unpersist`) when they are no longer needed is not always effective at addressing performance issues.

Consider another simple example of an iterative program in Figure 5.3. This is an abridged version of Spark's PageRank implementation which starts with `graph` as input and then iteratively refines it. (Note that the GraphX graph processing library in Spark represents graphs as two separate RDDs. We treat them here as a single RDD for simplicity.) At the start of an iteration on line 4, the graph from the previous iteration is persisted since it is used both on line 5 and line 7. As in our previous example, when an iteration has been completed, the graph from the previous iteration is unpersisted (line 9).

This pattern of caching is similar to the one from Figure 5.1c. However, we note a significant difference on line 8 of Figure 5.3. The call to `foreachPartition` serves to materialize `rankGraph`, that is, it forces the evaluation of the lazy transformations which define this graph. The reason this materialization is required is subtle. Recall that computation in Spark is lazy. In Figure 5.3 (unlike Figure 5.1c), there are no other actions within the loop. This would cause Spark to delay execution until the application reaches line 11. However, the persist annotations are *not* lazy, meaning the last annotation would take effect before the Spark job runs. In this case, the result would be that *no* caching takes place, which is certainly not what the developer intended. Our previous approach of caching before reuse and unpersisting when no further reuse occurs is useless without materialization!

The result of the action used to materialize the graph is not used by the application, but forcing evaluation places the graph in the cache as a side effect. This requires more knowledge of the details of Spark internals than should be necessary for an application developer and unnecessarily complicates the program. To see the impact on application performance we ran the Spark-Bench PageRank benchmark on a graph of 100,000 vertices

```
1  var rankGraph = graph.outerJoinVertices(...).map(...)

2  var iteration = 0
3  while (iteration < numIter) {
4    rankGraph.persist()

5    val rankUpdates = rankGraph.aggregateMessages(...)
6    prevRankGraph = rankGraph
7    rankGraph = rankGraph.outerJoinVertices(rankUpdates)
                          .persist()

     // Materialize the current graph and
     // unpersist the previous one
8    rankGraph.edges.foreachPartition(...)
9    prevRankGraph.unpersist()
10 }

11 rankGraph.vertices.values.sum()
```

Figure 5.3: Abridged version of PageRank in Spark

with and without the materialization on line 8. The full experimental setup is described in Section 5.4. The results are shown in Figure 5.4. With the materialization enabled, the cache is correctly populated and runtime increased linearly with the number of iterations. When materialization is disabled, the runtime grows quadratically since all previous iterations must be evaluated on each loop iteration. We will further examine some of these problems along with our automated solutions in the following sections.

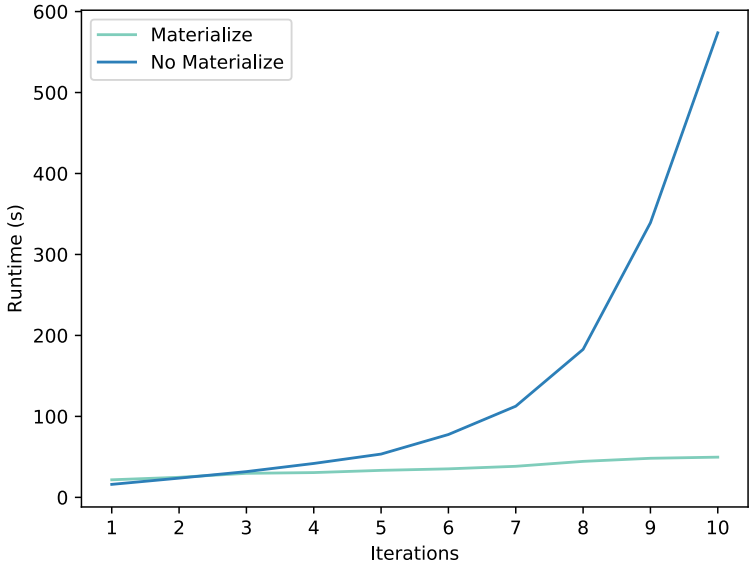


Figure 5.4: PageRank runtime with and without materialization

5.3 ReSpark

The following sections describe ReSpark, which aims to provide predictable performance to iterative Apache Spark programs with minimal effort on the part of the application developer. In particular, ReSpark eliminates the need for application developers to make manual persistence annotations in iterative applications. There are two main components to using ReSpark for improving application performance:

1. Programmers using ReSpark explicitly mark iterative constructs in the application program, exposing information about iteration to the ReSpark runtime.

2. At runtime, ReSpark makes use of the knowledge of the iterative program structure to automatically determine which RDDs to persist and when they can be unpersisted.

Section 5.3.1 explains how allowing programs to explicitly expose iterative patterns to the Spark runtime can improve application performance without the need to explicitly employ Spark's caching mechanisms. Section 5.3.2 describes how ReSpark automatically unpersists RDDs which are no longer needed without suffering from the performance problems described at the end of the previous section.

5.3.1 Explicit Iteration

In the example in Figure 5.3, the Spark program uses iteration to repeatedly update the RDD stored in the variable `next`. Because `next` is used twice in each iteration, it is persisted at the beginning of the loop to avoid recomputing the result when it is used on line 7. Caching is especially important for iterative computations since a cache miss could necessitate also recomputing the results of one or more previous iterations. We observe that if Spark were made aware of iteration inside application programs, it would be possible to identify these patterns and remove the need for manual caching annotations.

We formulate the problem of making appropriate caching decisions as a prediction problem. Specifically, we aim to enable Spark to learn what to persist given past application behaviour. When a new RDD is defined, we want to decide if the new RDD should be persisted. As we describe later, for each RDD we expose information on loops in the application program to the Spark runtime. This provides information about the loop and iteration in which each RDD was defined. One additional piece of information (which is already maintained by Spark) that is useful for this analysis is the *call site* of each RDD. The call site is simply a stack trace of the program which uniquely identifies the point in the application where the RDD is defined. This is useful since we expect that RDDs defined at the same call site will experience similar reuse patterns on successive iterations. An identifier for each loop, the current loop counter, and the call site forms what we refer to as the *loop context* of the RDD. In general, the input to this problem is a history of prior RDDs, their loop context, and actions that have been performed on these RDDs at the point a new RDD is defined.

Consider the sample Spark program in Figure 5.5. An outline of the dependency structure for RDDs in this program is given in Figure 5.6. Note that although call sites are represented as a stack trace by Spark, we simply use line numbers here since they are unambiguous for this example. RDD 1 represents the initial RDD on the first line of the program.

```
1  var next = sc.parallelize(1 to 100).map(i => (i,i))
2  var prev: RDD[(Int, Int)] = null
3  var n = 0

   // whileLoop below iterates from n=0..numIter
   // this is a macro defined by ReSpark
4  whileLoop(sc, {n += 1; n <= numIter}, {
5    val updates = next.map({ case (i, j) => (i, j + 1) })
6    prev = next
7    next = prev.join(updates)
8                .map({ case (i, (j, k)) => (i, j + k) })
9  })
10 next.count()
```

Figure 5.5: A Spark program making use of ReSpark's explicit iteration

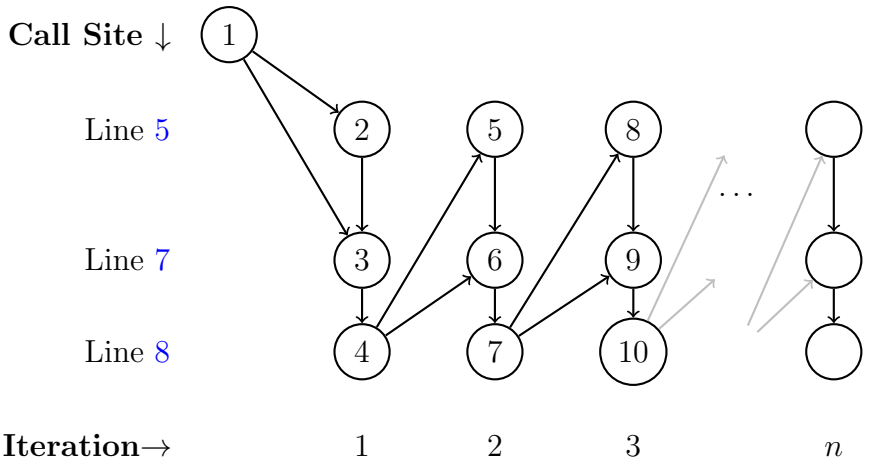


Figure 5.6: Generated RDDs for the job in Figure 5.5 labeled with RDD ID

For any technique making use of these annotations to be effective, we need to collect data on RDD reuse within the loop. On the first iteration of the loop, the call site information is not always useful since some RDDs referenced may be from call sites outside the loop when this may not be true on future iterations. Future iterations may instead make use of RDDs defined at call sites within the loop. For example, consider RDDs 2 and 3 in iteration 1 of Figure 5.6. Both depend on RDD 1, which was created on line 1, outside the loop. However, we see that RDDs 5 and 6 which are defined at the same call sites in the second iteration make use of an RDD defined on line 8. When the second iteration of the loop completes, we can examine all RDDs created in that iteration to determine the number of times RDDs generated at each call site were used. For example, the RDD generated on line 8 of Figure 5.5 of the first loop iteration will be used twice in the second iteration. In this case, ReSpark will predict that RDDs generated at line 8 in subsequent iterations will also be used twice. Thus, ReSpark will choose to persist those RDDs. For example, ReSpark will persist RDD 10 when it is generated on the third iteration.

In general, to determine when caching may be beneficial, we note that the cache will only be used if the RDD is used more than once. RDDs may be used in one of two ways: as input to a transformation to produce another RDD or in an action to compute a value from the RDD. By counting transformations and actions for an RDD, we can determine when reuse occurs for RDDs with a given loop context. Since we also have loop context information for each RDD, we can use this loop context to predict whether new RDDs with similar loop contexts will be reused, and to decide whether they should be persisted.

The example program in Figure 5.5 contains a macro named `whileLoop`, which is defined by ReSpark. The primary purpose of the `whileLoop` macro is to track what loop is currently being executed as well as count the current iteration of the loop. This information can be used to annotate RDDs with information describing which loop iteration generated each RDD. As we show later, the caching annotations present in the similar program described in Figure 5.3 are no longer necessary.

The macro requires three parameters. The first is a Spark "context" (distinct from our notion of loop context) which is used by all Spark programs to track shared state. The second is a test which determines when the loop will terminate. Finally, the macro is provided with the loop body to execute on each iteration. While we do not show this explicitly, we make use of iterators to implement similar functionality for the Spark Python API. For Java programs, we currently have developers explicitly mark the start and end of a loop as well as each iteration.

ReSpark uses a simple approach to solving the problem of predicting which RDDs will be reused. We add four additional pieces of metadata to each RDD:

- `unpersistPending` which flags an RDD marked for lazy unpersistence,
- `reuseCount` which specifies how many times an RDD is expected to be reused,
- and finally `loopId` and `loopIteration` which contain information on the loop where the RDD was defined (a unique reference to a specific invocation of ReSpark's `whileLoop` macro) and the value of the loop counter.

This additional data is set when ReSpark adds a persistence annotation to an RDD so that ReSpark can decide when the RDD will no longer be needed. We record the number of times an RDD is used from a single iteration as the `reuseCount`. An RDD is considered used either when it is an ancestor of another RDD or when it is used in an action.

When `reuseCount` is greater than one and we decide to persist an RDD, the count is treated as a hint for the number of times the RDD is expected to be used. That is, we expect that the number of uses of an RDD with a particular call site during one iteration is predictive of the number of uses RDDs defined at the same call site will have on future iterations. This enables ReSpark to decide what RDDs should be persisted and also when they are no longer needed and can be unpersisted (removed from the cache). Finally, we also persist RDDs that are defined outside of the loop if they are reused and unpersist them when the loop terminates. The full set of algorithms used to update usage information is given in Figure 5.7.

This assumption of similar reuse counts across iterations is key to ReSpark. Specifically, we currently assume that the uses of RDDs on the second iteration of a loop is predictive of the number of uses of RDDs at the same call sites on future iterations. This is currently a limitation of ReSpark that we intend to address in future work. However, this assumption is sufficient for many useful algorithms which we analyze in Section 5.4. To enable ReSpark to work with applications with changing reuse patterns, we expect to be able to use the same usage information we are currently collecting in concert with more advanced machine learning techniques to make more complex predictions.

```
loopRdds stores RDDs used in the current loop
outsideRdds stores RDDs used outside of all loops

Procedure OnRDDDefinition(rdd)
    if rdd is inside a loop then
        Store the current loop ID and iteration count to rdd
        if the current loop iteration is 2 then
            /* Record usage information */
            foreach dependency dep of rdd do
                if dep was created inside a loop then
                    Increment the use count for the call site of dep
                else
                    Persist dep and store it in outsideRdds
    else if call site of rdd has a use count greater than 1 then
        Persist rdd
        Set the flag rdd.unpersistPending
        Store the use count for the call site of rdd in reuseCount

Procedure OnLoopEnd()
    /* Unpersist any remaining RDDs outside the loop */
    foreach rdd in outsideRdds do
        Set the flag rdd.unpersistPending
        Store the use count for the call site of rdd in reuseCount

Procedure OnRDDAction(rdd)
    if the current loop has not been counted before then
        Increment the use count for the call site of rdd
```

Figure 5.7: Simplified algorithms for tracking RDD usage information

5.3.2 Lazy Unpersist

In addition to automatically determining what to persist, ReSpark also automatically determines when to unpersist RDDs which previously had persistence annotations added. ReSpark unpersists RDDs lazily, during evaluation, once RDDs have been used the predicted number of times. To do this, we make use of the `reuseCount` associated with each RDD. As mentioned previously, the `reuseCount` indicates how many times we expect an RDD to be reused. To decide when an RDD should be unpersisted, we simply need to track when each use occurs and decrement the `reuseCount`. When the `reuseCount` of an RDD reaches zero, then it can be unpersisted since we do not expect future use to occur. This approach avoids the need to explicitly materialize results as discussed in Section 5.2.2 since we ensure the Spark runtime does not unpersist an RDD when it is expected to be reused.

To track each use of an RDD so ReSpark can decrement the `reuseCount`, we tie each usage to an execution of a Spark *stage*. Stages are created by the Spark scheduler when an action executes to compute the result of actions or repartition data for transformations such as aggregation or joins. The algorithm used by ReSpark for discovering the association between stages and uses of an RDD is called on each stage scheduled to compute the result of evaluating an action on an RDD. ReSpark then traverses the lineage of the RDD associated with that stage. During this traversal, ReSpark records RDDs marked with the `unpersistPending` flag, which indicates RDDs previously received persistence annotations from ReSpark. These are RDDs which should be unpersisted once their `reuseCount` reaches zero.

The goal of this traversal algorithm is to discover which of these RDDs are used by each stage Spark has scheduled. ReSpark maintains two data structures to store the association between RDDs and stages: `waitingStages` and `waitingRdds`. `waitingStages` indicates which stages make use of an RDD. These are the stages which must complete for the RDD to be unpersisted. `waitingRdds` stores the reverse mapping: which RDDs are used by each stage. This indicates which RDDs should have their `reuseCount` decremented when a stage completes. Details of the algorithm used to populate these two structures given in Figure 5.8.

As mentioned earlier, Spark may schedule multiple stages to repartition intermediate results. These intermediate stages are referred to as *shuffle* stages. Since a shuffle stage can be expensive to recompute, Spark stores the output of shuffle stages to disk. Storing this output to disk means that if a descendant of an RDD which is reused is the result of a shuffle, we only need to wait for the shuffle to complete. This is because Spark can use the shuffle output stored on disk instead of recomputing the RDD. Exploiting this behaviour

```
Procedure OnResultStageScheduled(finalStage)
    waiting ← [(finalStage.rdd, finalStage.id)]
    Procedure visit(rdd, stageId)
        /* Record rdd to be unpersisted later */
        if rdd.unpersistPending then
            Add stageId to waitingStages for rdd
            Add rdd to waitingRdds for stageId

        /* Mark ancestors of this RDD to be visited */
        foreach dependency dep of rdd do
            if dep is a shuffle dependency then
                Add (dep, dep.stageId) to waiting
            else
                Add (dep, stageId) to waiting

    /* Visit all stages following the lineage of each RDD */
    visited ← ∅
    do
        Pop (rdd, stageId) from waiting
        if (rdd, stageId) ∉ visited then
            visit(rdd, stageId)
            Add (rdd, stageId) to visited
    until waiting is empty;
```

Figure 5.8: Finding required stages for an RDD to be unpersisted

of shuffle stages is an optimization which allows ReSpark to unpersist data which can be provided by Spark's shuffle output on disk.

Once the `waitingStages` and `waitingRdds` structures have been populated by the algorithm in Figure 5.8, we can use this information during job execution to decide when RDDs can be unpersisted. As each stage completes, the algorithm checks `waitingRdds` to see if there are any RDDs that may be ready to be unpersisted. If the stage which just completed corresponds to the last use of an RDD, then that RDD is unpersisted. This approach ensures that RDDs are persisted when requested and allows them to be unpersisted as soon when they are no longer needed. The full algorithm is shown in Figure 5.9.

Consider again Spark's PageRank implementation from Figure 5.3. Figure 5.10 shows RDDs generated in several iterations of the algorithm. Since a join is used to define both `rankUpdates` and `rankGraph`, each iteration results in two shuffle stages which repartition

```
Procedure OnStageFinish(stage)
    /* Check for any RDDs which can now be unpersisted */
    foreach rdd in waitingRdds for stage do
        Remove stage from the list of stages for rdd in waitingStages
        Decrement reuseCount for rdd
        if waitingStages for rdd is empty and reuseCount is zero then
            Unpersist rdd
```

Figure 5.9: Check for RDDs which can possibly be unpersisted

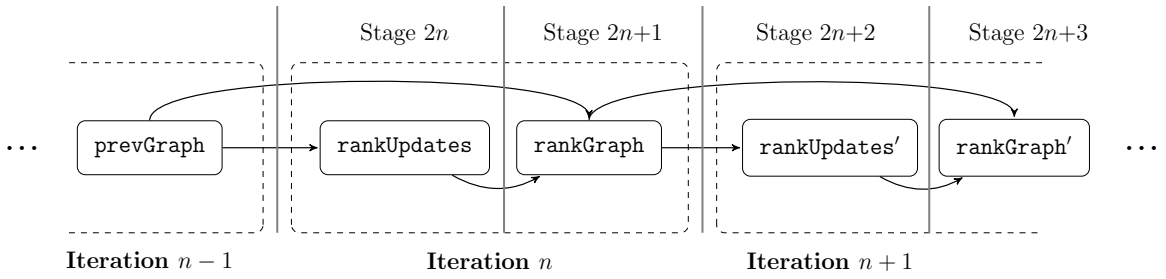


Figure 5.10: Several iterations of the Spark PageRank algorithm

data for these joins. The vertical lines in the diagram represent the division of each iteration into stages. The first stage of iteration n computes `rankUpdates` and the second computes `rankGraph`. Both of these stages depend on `prevRankGraph` which was persisted as shown in line 4 in Figure 5.3. We must consider which stages will use `prevRankGraph` to decide when it can be unpersisted.

The final stage of the program computes the result of the `sum` action on line 11. The ancestors of the iteratively computed `rankGraph` will then be visited in turn by the `visit` function in Figure 5.8. Consider what happens with this example program when the `visit` algorithm reaches stage $2n + 2$. Here the algorithm will visit `rankGraph` and see that the `unpersistPending` flag is set. As a shuffle boundary was crossed to reach `rankGraph` from `rankUpdates`, stage $2n + 2$ is marked as requiring completion before `rankGraph` can be unpersisted. Since `rankGraph` is also used by `rankGraph′`, it will be visited twice. Both `waitingRdds` and `waitingStages` will be updated as shown below to reflect the fact that stages $2n + 2$ and $2n + 3$ must complete before `rankGraph` can be unpersisted.

```
   waitingRdds: {"Stage 2n+2" => [rankGraph],
                 "Stage 2n+3" => [rankGraph], ...}
waitingStages: {rankGraph: ["Stage 2n+2", "Stage 2n+3"], ...}
```

After stage $2n + 2$ completes, the structures will be as follows:

```
  waitingRdds: {"Stage 2n+3" => [rankGraph], ...}
waitingStages: {rankGraph: ["Stage 2n+3"], ...}.
```

Finally, once stage $2n + 3$ completes, the entry for `rankGraph` in `waitingStages` will be empty and `rankGraph` can be unpersisted. Note that as shown in Figure 5.9, the `reuseCount` of `rankGraph` will also be checked before it is unpersisted. In this example, this has no effect on the outcome since `reuseCount` will initially be 2 and will be decremented to zero after stage $2n+3$. For applications with multiple actions, the check of `reuseCount` ensures that all actions have completed before RDDs are unpersisted. Without this check, stages which compute the results of these actions may not yet have been scheduled, which could cause RDDs to be unpersisted prematurely.

5.4 Evaluation

To evaluate the performance of our optimizations, we consider two scenarios. First, we examine Spark libraries which make use of iteration to show whether ReSpark is able to achieve comparable performance to library code annotated by Spark experts. We also consider two applications which make use of iterative jobs in Spark. A brief summary of the benchmarks used is given in Table 5.1. In both of these cases, we examine the runtime of the iterative jobs we identify both with and without our optimizations. When evaluating the algorithm with our optimizations, we remove any explicit persistence annotations and materialization added by the original application developer within the loop body.

We ran Spark on top of a cluster making use of the Hadoop distributed file system (HDFS) and Hadoop's YARN resource manager. We used three servers for each experiment, one for the Spark and HDFS NameNode, YARN ResourceManager, and Spark master. The other two hosted HDFS DataNodes, YARN NodeManagers and Spark executors. All experiments were performed with a pre-release of Spark 2.4.0 using YARN running on Hadoop 2.7.3. Each server has two six core Xeon E5-2620 processors operating at 2.10 GHz and 64 GB of memory. All figures report the average of three runs along with the minimum and maximum values.

5.4.1 Spark-Bench

We considered four algorithms which are distributed with Spark: shortest paths, PageRank, and strongly connected components (SCC), which are part of GraphX and K-means

Benchmark	Data-dependent termination	Loop Nesting	Materialization	Library
Shortest paths	✓			✓
PageRank			✓	✓
K-means	✓			✓
SCC		✓	✓	✓
LOPQ		✓		
BigITQ				

Table 5.1: Summary of evaluated Spark applications

clustering from MLlib. Benchmarks for each of these algorithms were obtained from version 2.0 of the Spark-Bench [84] benchmarking suite. The input data used for the benchmarks was produced by the random data generation facilities of Spark-Bench. The same settings for the Spark runtime were used for all the benchmarks run with Spark bench. Both the driver and each of two executors were allocated 4GB of memory. These specific algorithms were selected since they comprise all iterative algorithms within Spark-Bench. Note that we expect these algorithms to be highly optimized since they are implemented by the same team of developers working on the core of Spark.

We evaluated each Spark-Bench benchmark in against three versions of Spark:

1. Unmodified Spark with manual persistence annotations as well as explicit materialization of intermediate results.

2. ReSpark, which contains our modifications to support explicit iteration and lazy unpersistence. We also remove persistence annotations and materialization which exist inside iterative portions of the code.

3. Unmodified Spark, but with caching disabled. Note that we do not explicitly show results with caching disabled, but for all of the Spark-Bench benchmarks, we saw runtimes exceeding one hour.

We provide more a detailed performance evaluation for each algorithm in the following subsections. Since the code for these algorithms is verbose, we simply use a graphical representation that shows the structure of RDDs and iterations within the application

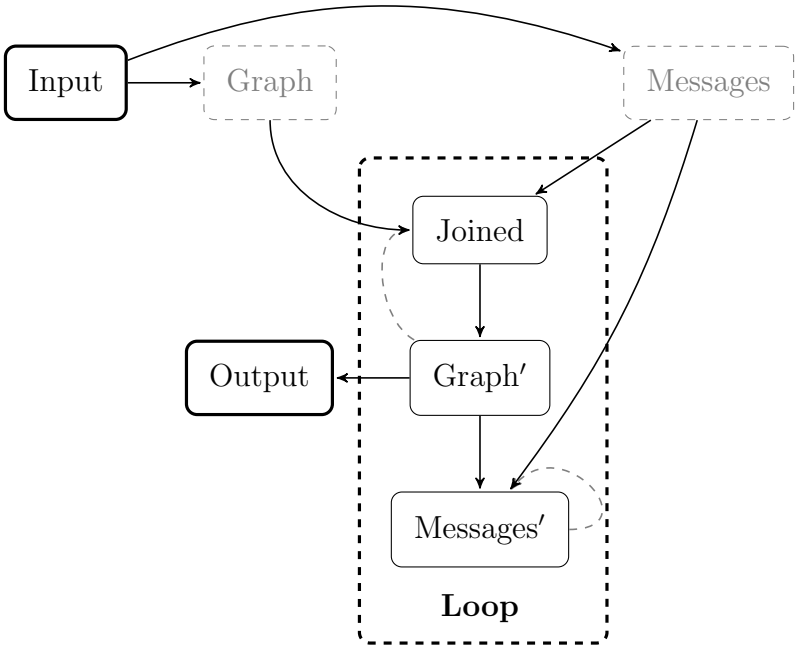


Figure 5.11: Structure of Spark's Pregel implementation

program. Each box represents the call site of an RDD in the program, with solid edges representing dependencies. RDDs surrounded by a dashed box represent the similarly-named RDD from a previous loop iteration.

Shortest paths

The shortest paths algorithm in Spark is part of the GraphX graph processing library which uses an approach similar to the Pregel [87] graph processing system. Spark-Bench searches for the shortest path between two vertices of a graph with log-normal degree distribution ($\mu = 4.0$ and $\sigma = 1.3$). Details on the graph generation are given in the description of the Pregel [87]. Spark's Pregel implementation simply iterates sending messages between vertices which update their local data. Only vertices which receive messages are permitted to send messages in subsequent rounds. Execution stops when all vertices have stopped sending messages. The shortest paths algorithm in Spark is formulated as a vertex program using Pregel. A simple overview of the structure of the Pregel implementation in Spark used by the shortest paths algorithm is given in Figure 5.11 Spark developers have decided to cache all RDDs at the **Graph** and **Messages** call sites on each iteration.

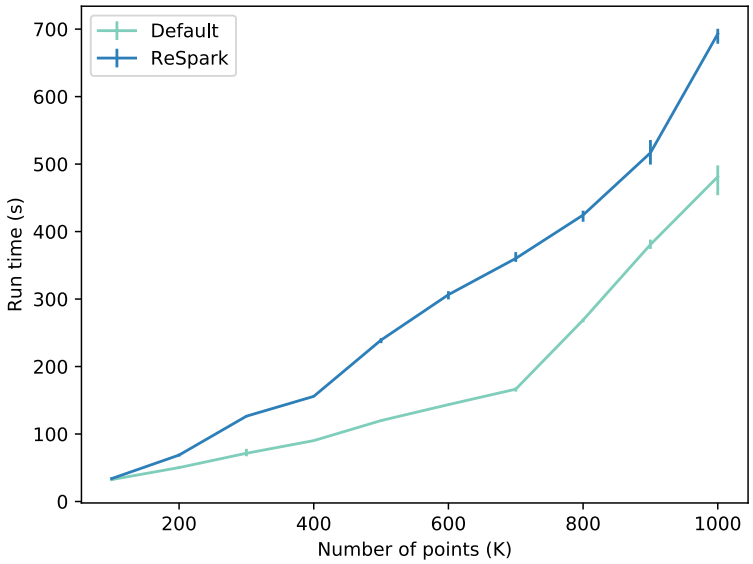


Figure 5.12: Shortest paths benchmark results

Spark implements the shortest paths algorithm according to the Pregel model by maintaining the length of the shortest path to the destination vertex. Each vertex sends a message to its neighbours containing this length with one step added for the extra vertex. Vertices will then update their local data to contain the minimum of the current length and the lengths received in these messages. Eventually these distances will converge so the source vertex contains the length of the shortest path to the destination. Figure 5.12 shows the runtime of the ReSpark and manually annotated versions of the algorithm for a graph with varying numbers of vertices. We see that ReSpark performs up to 116% slower than unmodified Spark. Without caching, shortest paths requires more than an hour to run, so ReSpark is able to obtain much of the potential benefit of caching. However, we found that the ReSpark version was approximately twice as slow as the manually annotated version in the worst case.

As expected, ReSpark does not persist RDDs generated on the first two iterations. However, in this case, performance is not significantly impacted since the shortest paths algorithm performs more than ten iterations with each iteration being relatively short. ReSpark also performs some extra caching on top of what is done by the manually annotated version of the algorithm. There are several instances within the GraphX library

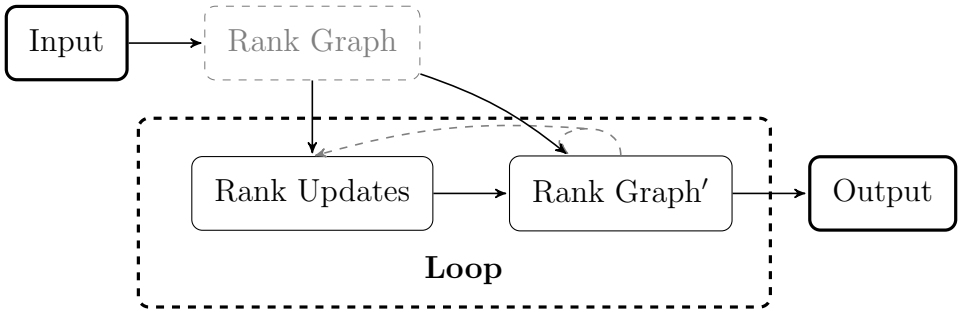


Figure 5.13: Structure of Spark's PageRank implementation

where ReSpark falsely infers reuse. A class within the Spark library simply copies instance variables (e.g. the value of manual persistence annotation) from an RDD `rdd1` to `rdd2`. ReSpark identifies this a second use of `rdd1` and places it in the cache. However, the `rdd2` does not depend on any of the data contained in `rdd1`. Therefore, caching `rdd1` will not will not reduce future computation time of `rdd2`. This caching is in fact harmful to performance since there is additional overhead for managing the cache when no actual reuse occurs.

PageRank

Spark's GraphX library also includes an implementation of the PageRank [102] algorithm. PageRank uses edges in a graph to calculate a rank for each node based on the incoming edges. The implementation of PageRank in Spark repeatedly sends messages to adjacent vertices and then performs a join of these messages with the current state of the graph to update the rank for each node. This algorithm runs a fixed number of iterations before terminating. An overview of the algorithm structure is given in Figure 5.13. Spark simply computes updates to the rank on each iteration and joins these updates with the original graph. To optimize this computation, GraphX in Spark persist RDDs generated during each iteration. The graph with the current rank of each node is then materialized the before the previous iteration is unpersisted to prevent the issues with unpersist discussed in Section 5.3.2.

We evaluate PageRank on graphs with a varying number of vertices generated the same way as the graphs for the shortest paths tests above. Without the explicit persistence annotations from the original GraphX implementation, ReSpark is able to achieve comparable performance, as shown in Figure 5.14. In the worst case (at 300 thousand points) ReSpark is still less than 10% slower than the version of PageRank with manual persistence annota-

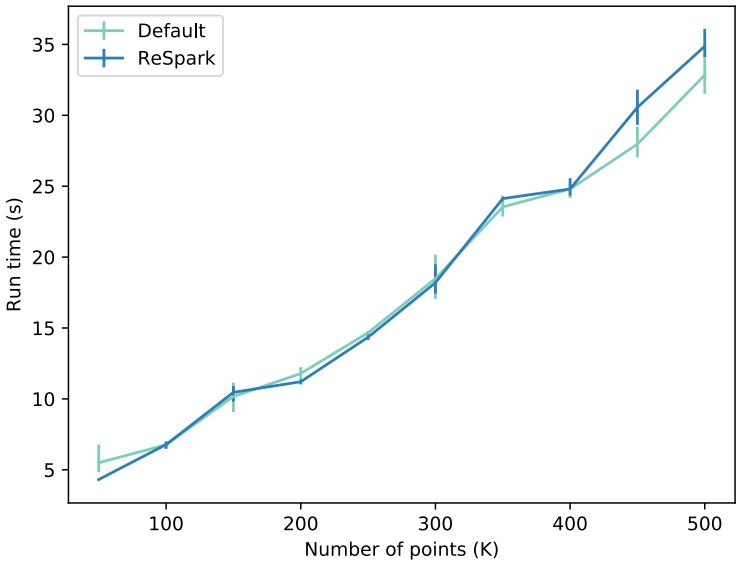


Figure 5.14: PageRank benchmark results

tions. Note that the widening performance gap for the 450 and 500 thousand point graphs is due to a larger percentage of the total runtime spent on the first two iterations which ReSpark does not cache.

K-means clustering

K-means in Spark is part of the MLlib machine learning package and uses the parallel k-means clustering algorithm [17]. After initializing cluster centres, the algorithm iterates over the dataset moving each cluster centre closer to its mean. Cluster centres are then recomputed until they converge or a maximum number of iterations (5) is reached. The structure of the algorithm is shown in Figure 5.15. RDDs generated at the **Costs** call site during cluster initialization are persisted, but the cluster initialization is a small fraction of the total run time. K-means clustering is a trivial example since reuse in the iterative computation is the initial dataset which is explicitly persisted. As shown in Figure 5.16, ReSpark achieves essentially identical performance to unmodified Spark, suggesting it introduces minimal overhead in these scenarios.

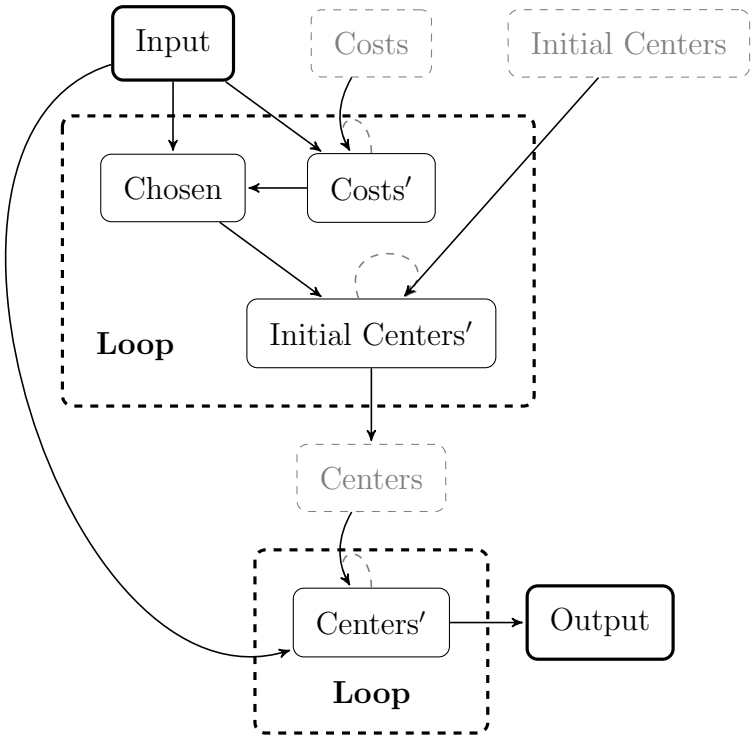


Figure 5.15: Structure of Spark's K-means implementation

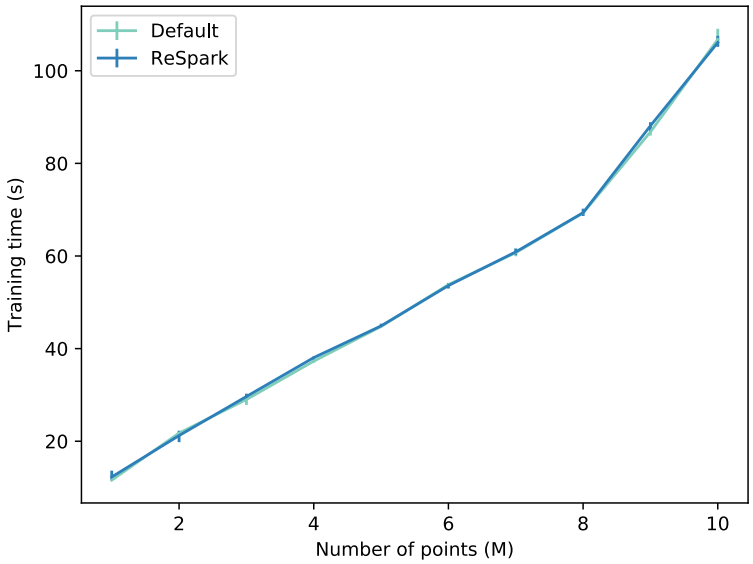


Figure 5.16: K-means clustering benchmark results

Strongly Connected Components

The strongly connected components algorithm aims to partition a graph into components in which there exists a path between every pair of vertices. Spark's SCC implementation first starts by assigning a unique component ID to each vertex and tagging each vertex as pending. The iterative portion consists of three stages. First, the graph is joined with the in and out degrees of each vertex to get all vertices which still have outgoing or incoming edges. Those vertices with no outgoing or incoming edges are marked as final and the component IDs are assigned in the final graph. The second stage uses Spark's Pregel implementation to collect the minimum component ID from all adjacent vertices. Finally, the third stage again uses Pregel to mark new vertices as final if there is a neighbour with the same colour which is marked as final. This continues until no unfinalized vertices remain or the configured number of iterations is complete. The structure of the algorithm including its connection with Pregel is shown in Figure 5.17. In addition to the caching which takes place inside Spark's Pregel implementation, RDDs generated at the **Graph** call site are also persisted. After materializing this graph, the graph from the previous iteration unpersisted.

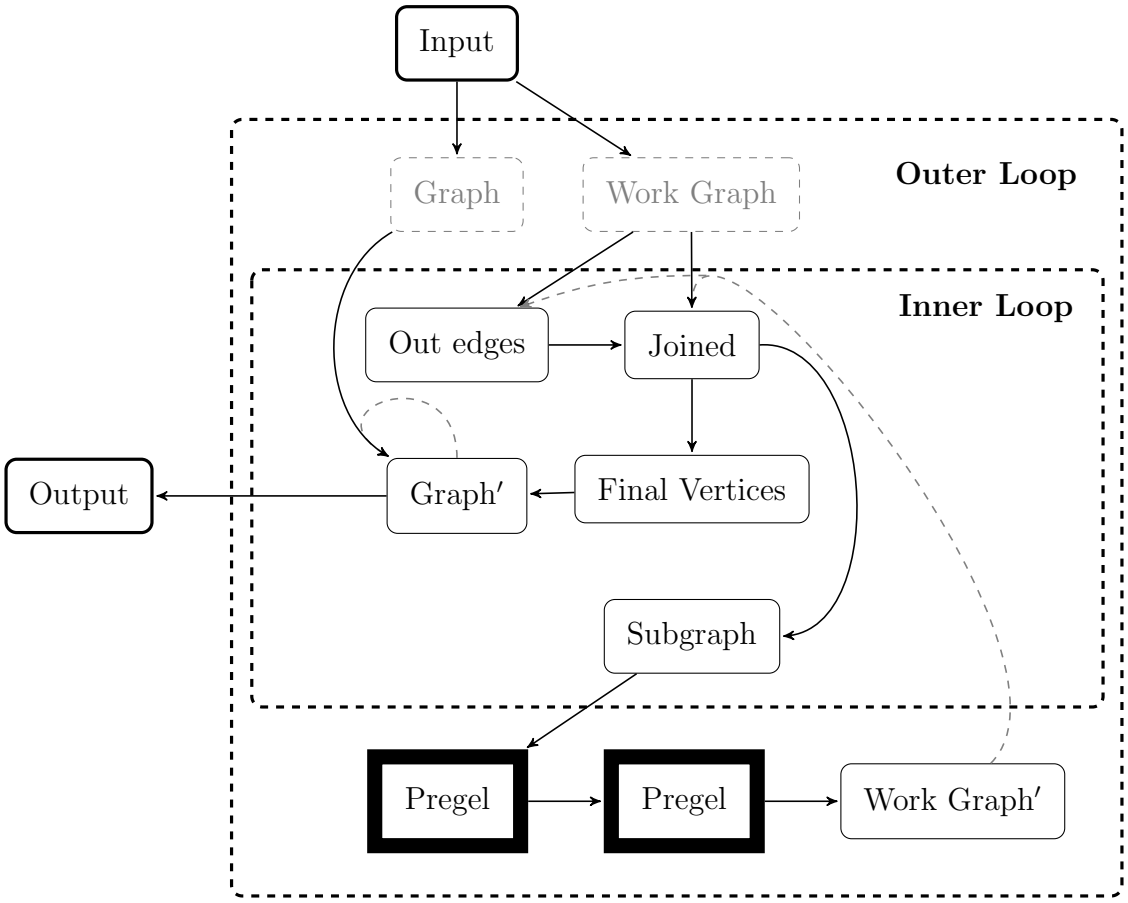


Figure 5.17: Structure of Spark's strongly connected components implementation (see Figure 5.11 for the Pregel implementation)

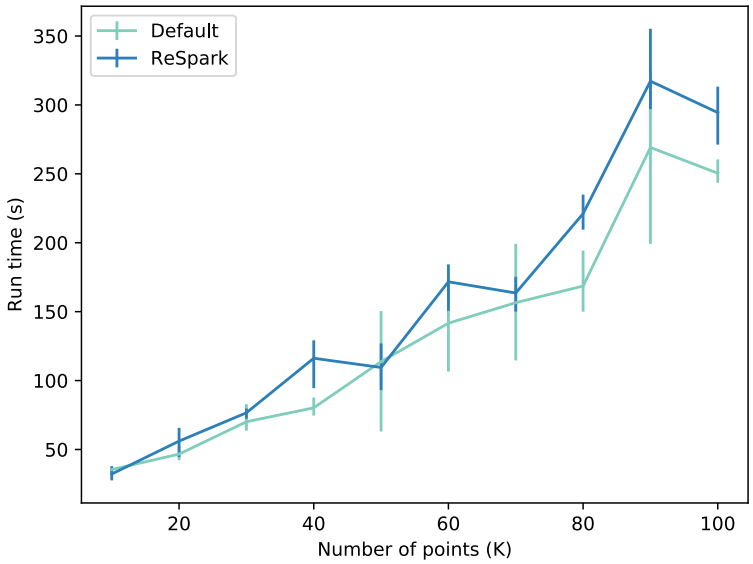


Figure 5.18: Strongly connected components benchmark results

We ran the SCC benchmark in Spark-Bench with its default of 3 iterations. The performance of ReSpark shown in Figure 5.18 is somewhat worse than unmodified Spark since ReSpark only obtains usage information for RDDs after two loop iterations have completed. However, there are many more iterations of the inner loops and usage information can be reused across iterations of the outer loop. ReSpark's automatic caching decisions result increase the runtime by less than 18% for the largest data set.

5.4.2 Other Iterative Algorithms

Approximate nearest neighbour

Nearest neighbour algorithms aim to provide the nearest vector in a given set to a query vector. Locally optimized product quantization (LOPQ) [68] is an approximate nearest neighbour algorithm which has been implemented by developers at Yahoo! using Spark [93]. LOPQ trains multiple K-means models on different splits of the data. The relevant code is given in Figure 5.19. We apply our iterative optimizations to the underlying K-means model as well as the iteration over the data splits. In addition, we remove all the provided

```
1 split_vecs.persist()
2 for split in xrange(M):
3   data = split_vecs.map(lambda x: x[split])
4   data.persist()
5   sub = KMeans.train(data, ...)
6   data.unpersist()
7   subquantizers.append(sub)
```

Figure 5.19: Iterative Python code for LOPQ in Spark

caching annotations (lines 1, 4, and 6 in Figure 5.19). Figure 5.20 shows the performance of ReSpark on the SIFT1M (2.3GB) and GIST1M (12GB) datasets [67]. ReSpark is able to identify the dataset which needs to be persisted on line 4 and achieve less than 1% slowdown compared to the original program with manual persistence allocations. In contrast, the version of the program with the manual persistence annotation disabled is 3.7% on the GIST1M dataset and 13.2% slower on the SIFT1M dataset. In this case, ReSpark was able to obtain almost the full benefit of caching without any manual persistence annotations.

Iterative quantization

Iterative quantization (ITQ) [62] is an algorithm for producing similarity-preserving binary codes for large image collections. The ITQ algorithm constructs a matrix representing these codes which is iteratively updated to produce the final result. We analyze BigITQ [58], a distributed Spark implementation of ITQ. A simplified excerpt of the code used for the algorithm is given in Figure 5.22. Results for the CIFAR-10 [78] dataset are given in Figure 5.21. In this case, we see that the algorithm actually benefits little from caching. The usage tracking done by ReSpark introduces a small overhead of approximately 3.2%.

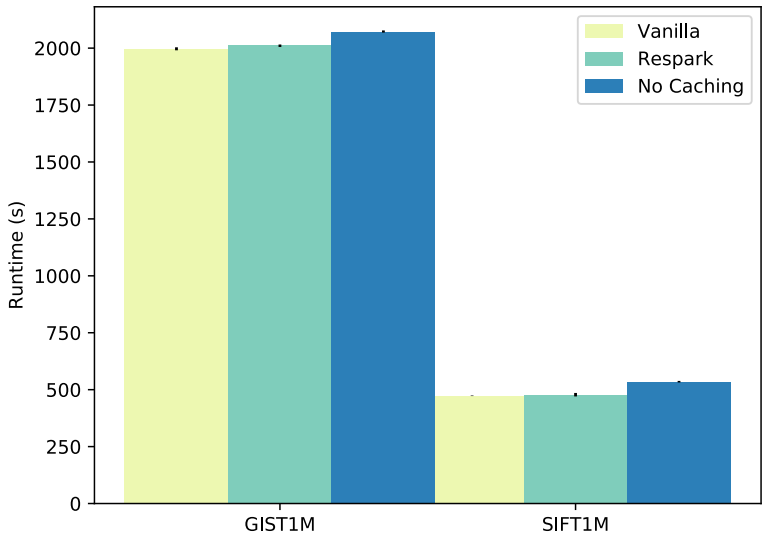


Figure 5.20: LOPQ performance using ReSpark

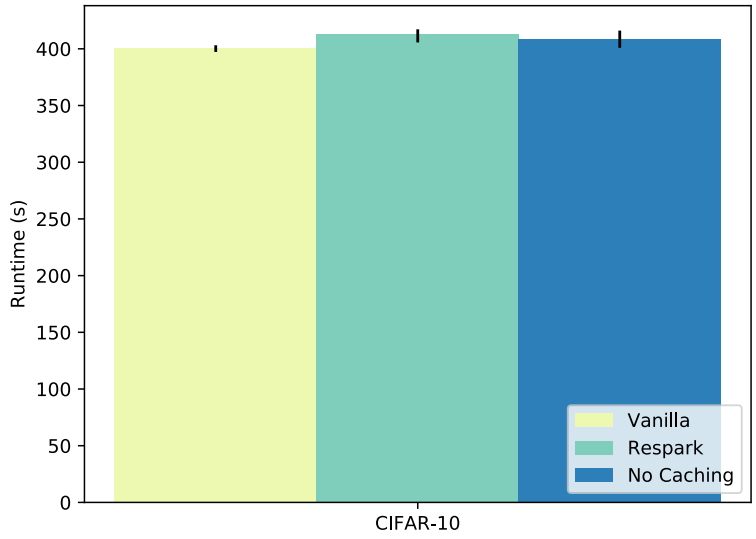


Figure 5.21: BigITQ performance using ReSpark

```
centered_data_idx.persist()

for iter_id in range(NITER):
  z = centered_data_idx.map(...)
  z.persist()
  c = z.join(centered_data_idx)....collect()
  ub, _, ua = np.linalg.svd(c[0][1])
  rot_matrix = np.array(ua.transpose()
                            .dot(ub.transpose()))
```

Figure 5.22: Iterative Python code for ITQ in Spark

5.5 Summary of ReSpark

We produced a set of modifications to the Apache Spark distributed computing framework to enable Spark to automatically optimize caching of intermediate results for applications using iteration. Our modifications, which refer to as ReSpark, allow programmers to expose information on iterative computation to the Spark runtime. ReSpark then automatically selects which intermediate results should be placed in the cache and when those results can be removed. In comparing these automated decisions with manual decisions made by expert Spark developers, we see that in many cases, ReSpark is able to obtain much of the benefit of caching without the need for expert knowledge. The key reasons for this are the simplicity of our prediction algorithm and the need for two iterations of a loop to complete before receiving any benefit from caching.

We expect that there are several opportunities to further introduce automatic optimizations for Spark. While ReSpark has demonstrated the effectiveness of automatically inferring cache annotations, it is still subject to Spark's least recently used cache eviction policy. A cache eviction policy which considers the cost of recomputation may be more effective. In addition, values such as the number of partitions used in an RDD or the number of executors also have an impact on the performance of Spark applications Like manual caching annotations, these values currently must be determined via expert knowledge of Spark and also trial and error. Unlike manual cache annotations, optimal values depend on the runtime environment.

Chapter 6

Conclusion and Future Work

In this thesis, we tackled challenges related to physical design for non-relational data systems. The tools and techniques presented aim to increase performance and usability for novice users of these systems.

In Chapter 3, we discussed the problem of schema design in NoSQL database systems. We showed that the current approaches rely on vague and self-contradictory rules of thumb. Our approach was to design an automated tool to perform the design task. As input, a user provides a conceptual model of data the applications wants to store along with expected queries and updates. Using this information, coupled with a cost model of the target system, our tool, the NoSQL Schema Evaluator (NoSE) produces a schema optimized for the given workload. In an evaluation based on an application benchmark, NoSE produced schemas outperformed manually designed schemas.

Chapter 4 presented an algorithm for producing a normalized logical model from a denormalized database design. This normalization removes redundancy implied by functional and inclusion dependencies from the data model. The resulting schema is in interaction-free inclusion dependency normal form. Our algorithm also produces a mapping between the normalized and denormalized designs which treats the denormalized design as a materialized view. This enables standard view-based query rewriting algorithms to answer queries written against the denormalized schema using the normalized schema. We showed how our algorithm was able to produce useful normalized designs given denormalized schemas from a variety of NoSQL database systems.

Finally, in Chapter 5 we furthered our efforts to increase the usability of non-relational data systems by examining the caching mechanisms used by Apache Spark. Similar to our examination of NoSQL systems, we found that there is no clear recommendation for when

data generated in Spark should be cached. Specifically, we studied the case of iterative programs where reuse is common and appropriate caching decisions have a significant impact on application performance. To solve the program of what data to cache, we formulated a prediction problem in which information from past loop iterations is used to predict future reuse, which can facilitate caching decisions. We proposed ReSpark, a set of enhancements to Spark which allow applications to explicitly express iterative computation. Using information on iteration present in the application program, we modify Spark to automatically make caching decisions. Experimental results show that our optimizations allow iterative Spark applications rewritten without caching annotations to achieve comparable performance to the original applications which contained caching decisions made by expert Spark users.

6.1 Future Work

There are many opportunities to extend and adapt this work to support other systems and use cases. Our schema design tool NoSE, has currently only been evaluated against the Apache Cassandra wide column store. We expect that similar techniques will benefit other NoSQL databases. NoSE coupled with our normalization algorithm presented in Chapter 4 form the foundation of an automated approach to schema management for NoSQL databases. We plan to explore how to use both of these tools in concert to enable applications to evolve their denormalized schema as requirements change. One important piece of this solution will be a method of efficiently migrating data from one design to another.

When working with existing designs, there are also other forms of redundancy which are not currently eliminated by our normalization algorithm. Furthermore, we would like to explore applying this algorithm in a data lake setting where it may be useful to identify relationships between data from multiple sources. We expect such an approach could be useful for data integration tasks such as schema and record matching.

Our work with ReSpark to simplify the development of high performance Spark applications identified other possible areas for improvement. For example, PrIter [130] showed a significant reduction in runtime for iterative MapReduce jobs by focusing computation on data which is most likely to lead to convergence. In examining issues Spark users experienced with caching, we also uncovered other problems whose solutions we expected can also be automated. For example, RDDs which have a long lineage in Spark should be checkpointed to reduce the overhead from analyzing ancestry each time an RDD is evaluated. Checkpointing stores the RDD to persistent storage and truncates its lineage.

Iterative jobs in Spark frequently generate RDDs with long lineages as the number of iterations increase, suggesting an automatic checkpointing mechanism would be useful here. Spark's default least recently used cache eviction policy may also not be optimal given that some RDDs are more expensive to compute. An eviction policy which takes into account the cost of recomputation may be more effective.

In addition, runtime parameters of Spark jobs must be carefully selected such as the number of partitions for an input dataset and the number of executors. However, optimal choices of these values can be difficult to determine and can vary depending on the dataset used. Other non-relational data systems such as Apache Flink or Twitter's Heron may also benefit from similar optimizations. Adding support to the Spark runtime for automatically tuning these values would provide similar benefits to those afforded by ReSpark.

References

[1] HBase: A distributed database for large datasets. Retrieved Jun. 14, 2018 from https://hbase.apache.org.

[2] Thinking in documents, April 2015. Retrieved Jun. 14, 2018 from https://www.mongodb.com/blog/post/thinking-documents-part-1.

[3] Amazon SimpleDB – Simple Database Service, 2018. Retrieved Jun. 14, 2018 from https://aws.amazon.com/simpledb.

[4] Apache Calcite, 2018. Retrieved Jun. 14, 2018 from https://calcite.apache.org.

[5] Apache CouchDB, 2018. Retrieved Jun. 14, 2018 from http://couchdb.apache.org.

[6] Apache Phoenix, 2018. Retrieved Jun. 14, 2018 from https://phoenix.apache.org.

[7] Kundera, 2018. Retrieved Jun. 14, 2018 from https://github.com/impetus-opensource/Kundera.

[8] MongoDB, 2018. Retrieved Jun. 14, 2018 from https://www.mongodb.com.

[9] Redis, 2018. Retrieved Jun. 14, 2018 from http://redis.io.

[10] Riak for Big Data Application Products, 2018. Retrieved Jun. 14, 2018 from http://basho.com/products.

[11] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB '00*, pages 496–505, Cairo, Egypt, 2000.

[12] Martin Andersson. Extracting an entity relationship schema from a relational database through reverse engineering. In *ER '94*, Lecture Notes in Computer Science, pages 403–419, Manchester, UK, Dec 1994.

[13] William Ward Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.

[14] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, 2016.

[15] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform Access to Non-relational Database Systems: The SOS Platform. In *Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 160–174. Springer Berlin Heidelberg, 2012.

[16] Antonio Badia and Daniel Lemire. A Call to Arms: Revisiting Database Design. *SIGMOD Rec.*, 40(3):61–69, November 2011.

[17] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. In *VLDB '12*, volume 5, pages 622–633, Istanbul, Turkey, March 2012.

[18] A. C. Bloesch and T. A. Halpin. ConQuer: A conceptual query language. In *Conceptual Modeling ER '96*, Lecture Notes in Computer Science, pages 121–133. Springer Berlin Heidelberg, January 1996.

[19] Nicolas Bruno and Surajit Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD '05*, Baltimore, MD, USA, 2005. Association for Computing Machinery, Inc.

[20] Nicolas Bruno and Rimma V. Nehme. Configuration-parametric Query Optimization for Physical Design Tuning. In *SIGMOD '08*, pages 941–952, Vancouver, BC, Canada, 2008.

[21] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *VLDB '10*, volume 3, pages 285–296, Singapore, Sep 2010.

[22] Francesca Bugiotti and Luca Cabibbo. A Comparison of Data Models and APIs of NoSQL Datastores. Technical report, Dipartamento di Ingegneria della Università di Roma Tre, 2013.

[23] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. Database Design for NoSQL Systems. In *Conceptual Modeling*, Lecture Notes in Computer Science, pages 223–231. Springer International Publishing, October 2014.

[24] Andre Calil and Ronaldo dos Santos Mello. SimpleSQL: A relational layer for SimpleDB. In *ADBIS '12*, pages 99–110, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] J. Calle, Y. Saez, and D. Cuadra. An evolutionary approach to the index selection problem. In *NaBIC '11*, pages 485–490, October 2011.

[26] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[27] Marco A. Casanova and Jose E. Amaral de Sa. Mapping uninterpreted schemes into entity-relationship diagrams: Two applications to conceptual schema design. *IBM Journal of Research and Development*, 28(1):82–94, Jan 1984.

[28] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, 1984.

[29] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[30] Emmanuel Cecchet et al. Performance and scalability of EJB applications. *ACM SIGPLAN Notices*, 37(11):246–261, 2002.

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[32] H. Chao-Qiang, Y. Shu-Qiang, T. Jian-Chao, and Y. Zhou. Rddshare: Reusing results of spark rdd. In *IEEE DSC 2016*, pages 370–375, Changsha, China, Jun 2016.

[33] Surajit Chaudhuri and Vivek Narasayya. AutoAdmin "What-if" Index Analysis Utility. In *SIGMOD '98*, pages 367–378, Seattle, WA, USA, 1998.

[34] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *ACM TODS*, 1(1):9–36, 1976.

[35] Alvin Cheung et al. Optimizing database-backed applications with query synthesis. In *PLDI '13*, pages 3–14, Seattle, WA, USA, 2013.

[36] M. Chevalier, M. El Malki, A. Kopliku, O. Teste, and R. Tournier. Implementation of multidimensional databases with document-oriented NoSQL. In *Big Data Analytics and Knowledge Discovery*, Lecture Notes in Computer Science, pages 379–390. Springer International Publishing, September 2015.

[37] Max Chevalier, Mohammed El Malki, Arlind Kopliku, Olivier Teste, and Ronan Tournier. Implantation not only SQL des bases de donnes multidimensionnelles. In *VSST, Grenade, Espagne*, 2015.

[38] Roger H. Chiang, Terence M. Barron, and Veda C. Storey. Reverse engineering of relational databases: Extraction of an EER model from a relational database. *Data & Knowledge Engineering*, 12(2):107–142, Mar 1994.

[39] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, May 2013.

[40] Sunil Choenni, Henk M. Blanken, and Thiel Chang. Index selection in relational databases. In *ICCI '93*, pages 491–496, Washington, DC, USA, 1993. IEEE Computer Society.

[41] Bobbie Cochrane. fAST refresh using mass query optimization. In *ICDE '01*, pages 391–, Heidelberg, Germany, 2001.

[42] Edgar F. Codd. Recent investigations into relational data base systems. Technical Report RJ1385, IBM, Apr 1974.

[43] Brian F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SOCC '10*, pages 143–154, Indianapolis, IN, USA, 2010.

[44] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB '04*, volume 30, pages 1098–1109, Toronto, ON, Canada, 2004.

[45] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. In *VLDB '11*, volume 4, pages 362–372, Seattle, WA, USA, March 2011.

[46] DataStax. About indexes in Cassandra, 2018. Retrieved Jun. 14, 2018 from https://docs.datastax.com/en/archived/cassandra/1.1/docs/ddl/indexes.html.

[47] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[48] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. Quartet: Harmonizing task scheduling and caching for cluster computing. In *HotStorage '16*, Santa Clara, CA, USA, 2016. USENIX Association.

[49] Ramez. Elmasri and Sham Navathe. *Fundamentals of database systems*. Addison-Wesley, Reading, Mass., 3rd ed. edition, 2000.

[50] E. Elnikety, T. Elsayed, and H. E. Ramadan. iHadoop: Asynchronous iterations for MapReduce. In *CloudCom '11*, pages 81–90, Athens, Greece, Nov 2011.

[51] Robert Escriva, Bernard Wong, and Emin Gn Sirer. HyperDex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, August 2012.

[52] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM TODS*, 2(3):262–278, Sep 1977.

[53] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.

[54] Eric Florenzano, Tyler Hobbs, Eric Evans, et al. Twissandra. Retrieved Jun. 14, 2018 from https://github.com/twissandra/twissandra.

[55] A. Gadkari, V.B. Nikam, and B.B. Meshram. Implementing Joins over HBase on Cloud Platform. In *CIT '14*, pages 547–554, September 2014.

[56] Hector Garcia-Molina et al. *Database systems: the complete book*. Pearson Prentice Hall, Upper Saddle River, N.J., 2 edition, 2009.

[57] Yuanzhen Geng, Xuanhua Shi, Cheng Pei, Hai Jin, and Wenbin Jiang. Lcs: An efficient data eviction strategy for spark. *International Journal of Parallel Programming*, pages 1–13, Nov 2016.

[58] Rohit Girdhar. Retrieved Jun. 14, 2018 from https://github.com/rohitgirdhar/BigITQ.

[59] Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *LICS '15*, pages 281–292, Kyoto, Japan, 2015. IEEE Computer Society.

[60] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, May 2001.

[61] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Data schema does matter, even in NoSQL systems! In *RCIS '16*, Grenoble, France, June 2016.

[62] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(12):2916–2929, Dec 2013.

[63] Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2018. Retrieved Jun. 14, 2018 from https://www.gurobi.com.

[64] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Sebastopol, CA, 2nd edition, 2011.

[65] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[66] Javier Luis Cánovas Izquierdo and Jordi Cabot. *Discovering Implicit Schemas in JSON Data*, pages 68–83. Springer, Berlin, Heidelberg, Jul 2013.

[67] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, January 2011.

[68] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2329–2336, June 2014.

[69] A. Kanade, A. Gopal, and S. Kanade. A study of normalization and embedding in MongoDB. In *IACC '14*, pages 416–421, Gurgaon, New Dehli, India, February 2014.

[70] Martti Kantola et al. Discovering functional and inclusion dependencies in relational databases. *Intl. Journal of Intelligent Systems*, 7(7):591–607, 1992.

[71] K. Kaur and R. Rani. Modeling and querying data in NoSQL databases. In *IEEE BigData 2013*, pages 1–7, Santa Clara, CA, USA, October 2013.

[72] Arthur M. Keller, Richard Jensen, and Shailesh Agarwal. Persistence software: Bridging object-oriented programming and relational databases. In *SIGMOD '93*, pages 523–528, Washington, DC, USA, 1993.

[73] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In *SIGMOD '92*, pages 393–402, San Diego, CA, USA, 1992.

[74] H. Kimura et al. CORADD: Correlation aware database designer for materialized views and indexes. In *VLDB '10*, pages 1103–1113, Singapore, 2010.

[75] Meike Klettke, Stefanie Scherzinger, and Uta Störl. Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In *BTW 2015*, Hamburg, Germany, 2015.

[76] Nitish Korla. Cassandra data modeling - practical considerations @ Netflix, 2013. Retrieved Jun. 14, 2018 from https://www.slideshare.net/nkorla1share/cass-summit-3.

[77] Piotr Koaczkowski and Henryk Rybiski. Automatic index selection in RDBMS by exploring query execution plan space. In *Advances in Data Management*, Studies in Computational Intelligence, pages 3–24. Springer Berlin Heidelberg, 2009.

[78] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[79] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[80] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-store 7 years later. In *VLDB '12*, volume 5, pages 1790–1801, Istanbul, Turkey, August 2012.

[81] Michael Lawley and Rodney Topor. A Query Language for EER Schemas. In *ADC '94*, pages 292–304, 1994.

[82] M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *IEEE TKDE*, 12(2):281–291, Mar 2000.

[83] Chongxin Li. Transforming relational database into HBase: A case study. In *ICSESS '10*, pages 683–687, Beijing, China, July 2010.

[84] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, 20(3):2575–2589, Sep 2017.

[85] Tok Wang Ling and Cheng Hian Goh. Logical database design with inclusion dependencies. In *ICDE '92*, pages 642–649, Tempe, AZ, USA, Feb 1992.

[86] Bin Liu, Wang-Pin Hsiung, J. Tatemura, and H. Hacigumus. SAGE: A logical and physical design tool for entity-group based new SQL systems. In *ICDE '14*, pages 1266–1269, Chicago, IL, USA, March 2014.

[87] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD '10*, pages 135–146, Indianapolis, IN, USA, 2010.

[88] Heikki Mannila and Kari-Jouko Räihä. Inclusion dependencies in database design. In *ICDE '86*, pages 713–718, Los Angeles, CA, USA, Feb 1986.

[89] Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *DKE*, 12(1):83–99, 1994.

[90] V. M. Markowitz and J. A. Makowsky. Identifying extended entity-relationship object structures in relational schemas. *IEEE Trans. on Software Engineering*, 16(8):777–790, Aug 1990.

[91] Christopher J. Matheus et al. Systems for knowledge discovery in databases. *IEEE Transactions on knowledge and data engineering*, 5(6):903–913, 1993.

[92] R Meersman. The RIDL conceptual language. Technical report, Research report, International Centre for Information Analysis Services, Control Data Belgium, Inc., Brussels, Belgium, 1982.

[93] Clayton Mellina and Marcel Kurovski. Retrieved Jun. 14, 2018 from https://github.com/yahoo/lopq.

[94] M. J. Mior and K. Salem. Renormalization of NoSQL database schemas. In *ER '18*, Xi'an, China, 2018. To appear.

[95] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. In *ICDE '16*, pages 181–192, Helsinki, Finland, May 2016.

[96] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. *IEEE TKDE*, 29(10):2275–2289, Oct 2017.

[97] Michael J. Mior. NoSE: Automated schema design for NoSQL applications, 2016. Retrieved Jun. 14, 2018 from https://github.com/michaelmior/NoSE.

[98] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, March 1992.

[99] John C. Mitchell. *Inference Rules for Functional and Inclusion Dependencies*, pages 58–69. PODS '83. ACM, 1983.

[100] Eric J Naiburg and Robert A Maksimchuck. *UML for database design*. Addison-Wesley Professional, 2001.

[101] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, Vancouver, BC, USA, 2008.

[102] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[103] Stratos Papadomanolakis and Anastassia Ailamaki. An integer linear programming approach to database design. *ICDEW*, pages 442–449, 2007.

[104] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. Efficient use of the query optimizer for automated physical design. In *VLDB '07*, pages 1093–1104, Vienna, Austria, 2007.

[105] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. In *VLDB '15*, volume 8, pages 774–785, Hawaii, USA, February 2015.

[106] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. In *EDBT '17*, pages 342–353, 2017.

[107] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In *ICDT '11*, pages 220–231, Uppsala, Sweden, 2011.

[108] Jay Patel. Cassandra data modeling best practices, part 1, 2012. Retrieved Jun. 14, 2018 from https://www.ebayinc.com/stories/blogs/tech/cassandra-data-modeling-best-practices-part-1/.

[109] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Commun. ACM*, 37(5):42–49, May 1994.

[110] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. In *VLDB '01*, volume 10, pages 334–350, Rome, Italy, Sep 2001.

[111] Alexander Rasin and Stan Zdonik. An Automatic Physical Design Tool for Clustered Column-stores. In *EDBT '13*, pages 203–214, Genoa, Italy, 2013. ACM.

[112] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[113] Stefanie Scherzinger, Eduardo Cunha De Almeida, Felipe Ickert, and Marcos Didonet Del Fabro. On the necessity of model checking NoSQL database schemas when building SaaS applications. In *TTC 2013*, pages 1–6, New York, NY, USA, 2013. ACM.

[114] Aaron Schram and Kenneth M. Anderson. MySQL to NoSQL: Data modeling challenges in supporting scalability. In *SPLASH '12*, pages 191–202, Tucson, AZ, USA, 2012. ACM.

[115] Alan Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE '00*, pages 101–110, Long Beach, CA, USA, 2000.

[116] Emily Stolfo. MongoDB schema design, 2013. Retrieved Jun. 14, 2018 from https://www.slideshare.net/mongodb/mongodb-schema-design-20356789.

[117] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *VLDB '05*, pages 553–564, Trondheim, Norway, 2005.

[118] Uta Störl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. Schemaless NoSQL data stores–object-NoSQL mappers to the rescue? In *Proc. BTW*, volume 15, 2015.

[119] Toby J. Teorey, Dongqing Yang, and James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv.*, 18(2):197–222, Jun 1986.

[120] A. H. M. ter Hofstede, H. A. Proper, and Th. P. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523, October 1993.

[121] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. In *VLDB '09*, volume 2, pages 1626–1629, Lyon, France, August 2009.

[122] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: a versatile tool for physical data independence. In *VLDB '96*, volume 5, pages 101–118, Bombay, India, April 1996.

[123] Tams Vajk, Lszl Dek, Krisztin Fekete, and Gergely Mezei. Automatic NoSQL schema development: A case study. In *Proceedings of the IASTED Multiconferences*. ACTA-PRESS, 2013.

[124] R. Varadarajan, V. Bharathan, A. Cary, J. Dave, and S. Bodagala. DBDesigner: A customizable physical design tool for Vertica analytic database. In *ICDE '14*, pages 1084–1095, Chicago, IL, USA, March 2014.

[125] Lanjun Wang et al. Schema management for document stores. In *VLDB '15*, volume 8, pages 922–933, Hawaii, USA, May 2015.

[126] Yue Wang, Yingzhong Xu, Yue Liu, Jian Chen, and Songlin Hu. QMapper for smart grid: Migrating SQL-based application to Hive. In *SIGMOD '15*, pages 647–658, Melbourne, Australia, 2015.

[127] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[128] Erci Xu, Mohit Saxena, and Lawrence Chiu. Neutrino: Revisiting memory caching for iterative data analytics. In *HotStorage 16*. USENIX Association, 2016.

[129] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[130] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC 11, pages 13:1–13:14, Cascais, Portugal, 2011. ACM.

[131] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68, Mar 2012.

[132] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB '04*, pages 1087–1097, Toronto, ON, Canada, 2004.

[133] D.C. Zilio, C. Zuzarte, S. Lightstone, Wenbin Ma, G.M. Lohman, R.J. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, and G. Valentin. Recommending materialized views and indexes with the IBM DB2 design advisor. In *International Conference on Autonomic Computing*, pages 180–187, May 2004.

APPENDICES

Appendix A

NoSE Workloads and Selected Schemas

This document contains a description of the workload and all the column families, query, and update plans used in the evaluation of NoSE. For the column family descriptions, underlined attributes denote attributes which are in the primary key of each column family. Attributes which are *italicized* are the clustering key attributes, with other primary key attributes forming the partition key. Any undecorated attributes are the values associated with each set of primary key values.

A.1 Conceptual Model

The conceptual model for RUBiS is given in the form of an entity graph in Figure A.1. Note that we have chosen to omit the labels for relationships.

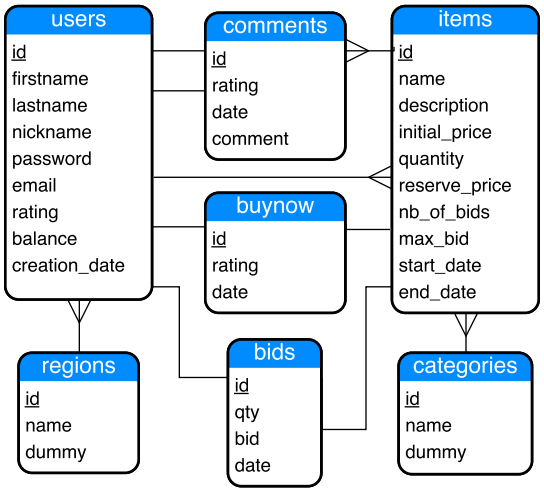


Figure A.1: Entity graph for the RUBiS benchmark

A.2 Workload

Sets of statements in the workload are combined into *interactions* as described in 3.8. The statements used as input to NoSE along with the interactions are given below. Specific frequencies for each interaction are given in Section A.3 when the schemas are described.

BrowseCategories

```
SELECT users.nickname, users.password FROM users WHERE users.id = ?
SELECT categories.id, categories.name FROM categories WHERE categories.dummy
= 1
```

ViewBidHistory

```
SELECT items.name FROM items WHERE items.id = ?
SELECT users.id, users.nickname, bids.id, item.id, bids.qty, bids.bid, bids.date
FROM users.bids.item WHERE item.id = ?  ORDER BY bids.date
```

ViewItem

```
SELECT items.* FROM items WHERE items.id = ?
SELECT bids.* FROM items.bids WHERE items.id = ?
```

SearchItemsByCategory

```
SELECT items.id, items.name, items.initial_price, items.max_bid, items.nb_of_-
bids, items.end_date FROM items.category WHERE category.id = ?  AND items.end_-
date >= ?  LIMIT 25
```

ViewUserInfo

```
SELECT users.* FROM users WHERE users.id = ?
SELECT comments.id, comments.rating, comments.date, comments.comment
FROM comments.to_user WHERE to_user.id = ?
```

RegisterItem

```
INSERT INTO items SET id=?, name=?, description=?, initial_price=?, quantity=?,
reserve_price=?, buy_now=?, nb_of_bids=0, max_bid=0, start_date=?, end_date=?
AND CONNECT TO category(?), seller(?)
```

RegisterUser

```
INSERT INTO users SET id=?, firstname=?, lastname=?, nickname=?, password=?,
email=?, rating=0, balance=0, creation_date=?  AND CONNECT TO region(?)
```

BuyNow

```
SELECT users.nickname FROM users WHERE users.id=?
SELECT items.* FROM items WHERE items.id=?
```

StoreBuyNow

```
SELECT items.quantity, items.nb_of_bids, items.end_date FROM items WHERE items.id=?
UPDATE items SET quantity=?, nb_of_bids=?, end_date=?  WHERE items.id=?
INSERT INTO buynow SET id=?, qty=?, date=?  AND CONNECT TO item(?), buyer(?)
```

PutBid

```
SELECT users.nickname, users.password FROM users WHERE users.id=?
SELECT items.* FROM items WHERE items.id=?
SELECT bids.qty, bids.date FROM bids.item WHERE item.id=?  ORDER BY bids.bid
LIMIT 2
```

StoreBid

```
INSERT INTO bids SET id=?, qty=?, bid=?, date=?  AND CONNECT TO item(?), user(?)
SELECT items.nb_of_bids, items.max_bid FROM items WHERE items.id=?
UPDATE items SET nb_of_bids=?, max_bid=?  WHERE items.id=?
```

PutComment

```
SELECT users.nickname, users.password FROM users WHERE users.id=?
SELECT items.* FROM items WHERE items.id=?
SELECT users.* FROM users WHERE users.id=?
```

StoreComment

```
SELECT users.rating FROM users WHERE users.id=?
UPDATE users SET rating=?  WHERE users.id=?
INSERT INTO comments SET id=?, rating=?, date=?, comment=?  AND CONNECT TO to_-
user(?), from_user(?), item(?)
```

AboutMe

```
SELECT users.* FROM users WHERE users.id=?
SELECT comments_received.* FROM users.comments_received WHERE users.id = ?
SELECT from_user.nickname FROM comments.from_user WHERE comments.id = ?
SELECT bought_now.*, items.* FROM items.bought_now.buyer WHERE buyer.id = ?
AND bought_now.date>=?
SELECT items.* FROM items.seller WHERE seller.id=?  AND items.end_date >=?
SELECT items.* FROM items.bids.user WHERE user.id=?  AND items.end_date>=?
```

SearchItemsByRegion

```
SELECT items.id, items.name, items.initial_price, items.max_bid, items.nb_of_-
bids, items.end_date FROM items.seller WHERE seller.region.id = ?
AND items.category.id = ?  AND items.end_date >= ?  LIMIT 25
```

BrowseRegions

```
SELECT regions.id, regions.name FROM regions WHERE regions.dummy = 1
```

A.3 Schemas

A.3.1 NoSE Bidding

Interaction frequencies

7.65	BrowseCategories
1.54	ViewBidHistory
14.17	ViewItem
15.94	SearchItemsByCategory
2.48	ViewUserInfo
0.53	RegisterItem
1.07	RegisterUser
1.16	BuyNow
1.10	StoreBuyNow
5.40	PutBid
3.74	StoreBid
0.46	PutComment
0.45	StoreComment
1.71	AboutMe
6.34	SearchItemsByRegion
5.39	BrowseRegions

Column Families

i3722443462
categories.dummy, *categories.id*, categories.name

i193173044
items.id, *bids.date*, *bids.id*, *users.id*, users.nickname, bids.qty, bids.bid

i1888493477
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3264766123
items.id, *categories.id*

i3392968797
categories.id, *items.end_date*, *items.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids

i2906147889
users.id, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date

i3157175159
users.id, *comments.id*, comments.rating, comments.date, comments.comment

i3563903410
items.id, *bids.bid*, *bids.id*, bids.qty, bids.date

i915430138
comments.id, *users.id*, users.nickname

i2578518014
items.id, *buynow.id*, *buynow.date*, *users.id*

i2653317939
users.id, *buynow.date*, *buynow.id*, *items.id*, buynow.qty, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i578710568
items.id, *users.id*

i2337785568
users.id, *items.id*, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3553045793
users.id, *items.end_date*, *bids.id*, *items.id*

i3639792234
regions.id, *categories.id*, *users.id*, *items.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids, items.end_date

i1683742356
users.id, *regions.id*

i1912786220
items.id, *users.id*, *regions.id*

i590232953
regions.dummy, *regions.id*, regions.name

Plans

BrowseCategories
Get **i2906147889**
Get **i3722443462**

ViewBidHistory
Get **i1888493477**
Get **i193173044**

ViewItem
Get **i1888493477**
Get **i3563903410**

SearchItemsByCategory
Get **i3392968797**

ViewUserInfo
Get **i2906147889**
Get **i3157175159**

BuyNow
Get **i2906147889**
Get **i1888493477**

StoreBuyNow
Get **i1888493477**
Insert into **i1888493477**
Get **i1888493477**, Get **i3264766123**, Delete from **i3392968797**, Insert into **i3392968797**
Get **i2578518014**, Insert into **i2653317939**
Get **i578710568**, Insert into **i2337785568**
Get **i193173044**, Delete from **i3553045793**, Insert into **i3553045793**
Get **i1912786220**, Get **i3264766123**, Insert into **i3639792234**
Insert into **i2578518014**
Get **i1888493477**, Insert into **i2653317939**

PutBid
Get **i2906147889**
Get **i1888493477**
Get **i3563903410**

StoreBid
Get **i1888493477**
Get **i2906147889**, Insert into **i193173044**
Insert into **i3563903410**
Get **i1888493477**, Insert into **i3553045793**
Insert into **i1888493477**
Get **i1888493477**, Get **i3264766123**, Insert into **i3392968797**
Get **i2578518014**, Insert into **i2653317939**
Get **i578710568**, Insert into **i2337785568**
Get **i1912786220**, Get **i3264766123**, Insert into **i3639792234**

PutComment
Get **i2906147889**
Get **i1888493477**
Get **i2906147889**

StoreComment
Get **i2906147889**
Insert into **i2906147889**
Insert into **i3157175159**
Get **i2906147889**, Insert into **i915430138**

AboutMe
Get **i2906147889**
Get **i3157175159**
Get **i915430138**
Get **i2653317939**
Get **i2337785568**, Filter by items.end_date
Get **i3553045793**, Get **i1888493477**

SearchItemsByRegion
Get **i3639792234**, Filter by items.end_date

BrowseRegions
Get **i590232953**

RegisterItem
Insert into **i1888493477**
Insert into **i3264766123**
Insert into **i3392968797**
Insert into **i578710568**
Insert into **i2337785568**
Get **i1683742356**, Insert into **i3639792234**
Get **i1683742356**, Insert into **i1912786220**

RegisterUser
Insert into **i2906147889**
Insert into **i1683742356**

A.3.2 Normalized

Column Families

categories
categories.id, categories.name, categories.dummy

regions
regions.id, regions.name, regions.dummy

items
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

comments
comments.id, comments.rating, comments.date, comments.comment

users_by_region
regions.id, *users.id*, users.nickname

users
users.id, *regions.id*, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date

bids
bids.id, *users.id*, *items.id*, bids.qty, bids.bid, bids.date

buynow
buynow.id, *items.id*, buynow.qty, buynow.date

all_categories
categories.dummy, *categories.id*

all_regions
regions.dummy, *regions.id*

bids_by_item
items.id, *bids.id*

items_by_category
categories.id, *items.end_date*, *items.id*

items_by_region
regions.id, *categories.id*, *items.end_date*, *items.id*, *users.id*

comments_by_user
users.id, *comments.id*

user_items_sold
users.id, *items.end_date*, *items.id*

buynow_by_user
users.id, *buynow.date*, *buynow.id*

bids_by_user
users.id, *bids.date*, *bids.id*

Plans

BrowseCategories
Get **users**
Get **all_categories**, Get **categories**

ViewBidHistory
Get **items**
Get **bids_by_item**, Get **bids**, Get **users**

ViewItem
Get **items**
Get **bids_by_item**, Get **bids**

SearchItemsByCategory
Get **items_by_category**, Get **items**

SearchItemsByRegion
Get **items_by_region**, Get **items**

BrowseRegions
Get **all_regions**, Get **regions**

ViewUserInfo
Get **users**, Get **regions**
Get **comments_by_user**, Get **comments**

BuyNow
Get **users**
Get **items**

PutBid
Get **users**
Get **items**
Get **bids_by_item**, Get **bids**

PutComment
Get **users**
Get **items**
Get **users**

AboutMe
Get **users**
Get **comments_by_user**, Get **comments**
Get **buynow_by_user**, Get **buynow**, Get **items**
Get **user_items_sold**, Get **items**
Get **bids_by_user**, Get **bids**, Get **items**

RegisterItem
Insert into **items**
Insert into **user_items_sold**
Insert into **items_by_category**
Get **users**, Insert into **items_by_region**

RegisterUser
Get **regions**, Insert into **users**
Insert into **users_by_region**

StoreBuyNow
Get **items**, Insert into **items**
Insert into **buynow**
Insert into **buynow_by_user**

StoreBid
Get **items**, Insert into **items**
Insert into **bids**
Insert into **bids_by_item**
Insert into **bids_by_user**

StoreComment
Get **users**, Insert into **users**
Insert into **comments**
Insert into **comments_by_user**

A.3.3 Expert

Column Families

users_by_region
regions.id, *users.id*, users.nickname

user_data
users.id, *regions.id*, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date, regions.name

user_buynow
users.id, *buynow.date*, *buynow.id*, *items.id*, buynow.qty

user_items_bid_on
users.id, *items.end_date*, *bids.id*, *items.id*, bids.qty

user_items_sold
users.id, *items.end_date*, *items.id*

user_comments_received
users.id, *comments.id*, *items.id*, comments.rating, comments.date, comments.comment

commenter
comments.id, *users.id*, users.nickname

items_with_category
items.id, *categories.id*, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

item_bids
items.id, *bids.id*, *users.id*, items.max_bid, users.nickname, bids.qty, bids.bid, bids.date

items_by_category
categories.id, *items.end_date*, *items.id*

category_list
categories.dummy, *categories.id*, categories.name

region_list
regions.dummy, *regions.id*, regions.name

regions
regions.id, regions.name

Plans

BrowseCategories
Get **user_data**
Get **category_list**

ViewBidHistory
Get **items_with_category**
Get **item_bids**

ViewItem
Get **items_with_category**
Get **item_bids**

SearchItemsByCategory
Get **items_by_category**, Get **items_with_category**

SearchItemsByRegion
Get **users_by_region**
Get **items_by_category**, Get **items_with_category**

BrowseRegions
Get **region_list**, Get **regions**

ViewUserInfo
Get **user_data**
Get **user_comments_received**, Get **commenter**

BuyNow
Get **user_data**
Get **items_with_category**

PutBid
Get **user_data**
Get **items_with_category**
Get **item_bids**

PutComment
Get **user_data**
Get **items_with_category**
Get **user_data**

AboutMe
Get **user_data**
Get **user_comments_received**, Get **commenter**
Get **user_buynow**, Get **items_with_category**
Get **user_items_sold**, Get **items_with_category**
Get **user_items_bid_on**, Get **items_with_category**

RegisterItem
Insert into **items_with_category**
Insert into **user_items_sold**
Insert into **items_by_category**

RegisterUser
Get **regions**, Insert into **user_data**
Insert into **users_by_region**

StoreBuyNow
Get **items_with_category**, Insert into **items_with_category**, Delete from **items_by_category**, Insert into **items_by_category**
Insert into **user_buynow**

StoreBid
Get **item_bids**, Insert into **item_bids**
Get **items_with_category**, Insert into **items_with_category**
Insert into **user_items_bid_on**

StoreComment
Get **user_data**, Insert into **user_data**
Insert into **user_comments_received**

A.3.4 NoSE Browsing

Interaction frequencies

4.44	BrowseCategories
0	ViewBidHistory
22.95	ViewItem
27.77	SearchItemsByCategory
4.41	ViewUserInfo
0	RegisterItem
0	RegisterUser
0	BuyNow
0	StoreBuyNow
0	PutBid
0	StoreBid
0	PutComment
0	StoreComment
0	AboutMe
8.26	SearchItemsByRegion
3.21	BrowseRegions

Column Families

i3722443462
categories.dummy, *categories.id*, categories.name

i193173044
items.id, *bids.date*, *bids.id*, *users.id*, users.nickname, bids.qty, bids.bid

i1888493477
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3220017915
items.id, *bids.id*, bids.qty, bids.bid, bids.date

i3392968797
categories.id, *items.end_date*, *items.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids

i2906147889
users.id, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date

i3157175159
users.id, *comments.id*, comments.rating, comments.date, comments.comment

i4047225742
regions.id, *categories.id*, *items.end_date*, *items.id*, *users.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids

i590232953
regions.dummy, *regions.id*, regions.name

Plans

BrowseCategories
Get **i2906147889**
Get **i3722443462**

ViewBidHistory
Get **i1888493477**
Get **i193173044**

ViewItem
Get **i1888493477**
Get **i3220017915**

SearchItemsByCategory
Get **i3392968797**

ViewUserInfo
Get **i2906147889**
Get **i3157175159**

SearchItemsByRegion
Get **i4047225742**

BrowseRegions
Get **i590232953**

A.3.5 NoSE 10×

Interaction frequencies

7.65	BrowseCategories
1.54	ViewBidHistory
14.17	ViewItem
15.94	SearchItemsByCategory
2.48	ViewUserInfo
5.3	RegisterItem
10.7	RegisterUser
1.16	BuyNow
11.0	StoreBuyNow
5.40	PutBid
37.4	StoreBid
0.46	PutComment
4.5	StoreComment
1.71	AboutMe
6.34	SearchItemsByRegion
5.39	BrowseRegions

Column Families

i3722443462
categories.dummy, *categories.id*, categories.name

i193173044
items.id, *bids.date*, *bids.id*, *users.id*, users.nickname, bids.qty, bids.bid

i1888493477
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3264766123
items.id, *categories.id*

i1175133162
categories.id, *items.end_date*, *items.id*

i2906147889
users.id, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date

i3157175159
users.id, *comments.id*, comments.rating, comments.date, comments.comment

i3563903410
items.id, *bids.bid*, *bids.id*, bids.qty, bids.date

i3128537325
comments.id, *users.id*

i2578518014
items.id, *buynow.id*, *buynow.date*, *users.id*

i2653317939
users.id, *buynow.date*, *buynow.id*, *items.id*, buynow.qty, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i578710568
items.id, *users.id*

i2337785568
users.id, *items.id*, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3553045793
users.id, *items.end_date*, *bids.id*, *items.id*

i184475158
categories.id, *regions.id*, *users.id*, *items.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids, items.end_date

i1683742356
users.id, *regions.id*

i101364659
items.id, *regions.id*, *users.id*

i590232953
regions.dummy, *regions.id*, regions.name

Plans

BrowseCategories
Get **i2906147889**
Get **i3722443462**

ViewBidHistory
Get **i1888493477**
Get **i193173044**

ViewItem
Get **i1888493477**
Get **i3563903410**

SearchItemsByCategory
Get **i1175133162**, Get **i1888493477**

ViewUserInfo
Get **i2906147889**
Get **i3157175159**

BuyNow
Get **i2906147889**
Get **i1888493477**

StoreBuyNow
Get **i1888493477**
Insert into **i1888493477**
Get **i3264766123**, Delete from **i1175133162**, Insert into **i1175133162**
Get **i2578518014**, Insert into **i2653317939**
Get **i578710568**, Insert into **i2337785568**
Get **i193173044**, Delete from **i3553045793**, Insert into **i3553045793**
Get **i101364659**, Get **i3264766123**, Insert into **i184475158**
Insert into **i2578518014**
Get **i1888493477**, Insert into **i2653317939**

PutBid
Get **i2906147889**
Get **i1888493477**
Get **i3563903410**

StoreBid
Get **i1888493477**
Get **i2906147889**, Insert into **i193173044**
Insert into **i3563903410**
Get **i1888493477**, Insert into **i3553045793**
Insert into **i1888493477**
Get **i2578518014**, Insert into **i2653317939**
Get **i578710568**, Insert into **i2337785568**
Get **i101364659**, Get **i3264766123**, Insert into **i184475158**

PutComment
Get **i2906147889**
Get **i1888493477**
Get **i2906147889**

StoreComment
Get **i2906147889**
Insert into **i2906147889**
Insert into **i3157175159**
Insert into **i3128537325**

AboutMe
Get **i2906147889**
Get **i3157175159**
Get **i3128537325**, Get **i2906147889**
Get **i2653317939**
Get **i2337785568**, Filter by items.end_date
Get **i3553045793**, Get **i1888493477**

SearchItemsByRegion
Get **i184475158**, Filter by items.end_date

BrowseRegions
Get **i590232953**

RegisterItem
Insert into **i1888493477**
Insert into **i3264766123**
Insert into **i1175133162**
Insert into **i578710568**
Insert into **i2337785568**
Get **i1683742356**, Insert into **i184475158**
Get **i1683742356**, Insert into **i101364659**

RegisterUser
Insert into **i2906147889**
Insert into **i1683742356**

A.3.6 NoSE 100×

Interaction frequencies

7.65	BrowseCategories
1.54	ViewBidHistory
14.17	ViewItem
15.94	SearchItemsByCategory
2.48	ViewUserInfo
53	RegisterItem
107	RegisterUser
1.16	BuyNow
110	StoreBuyNow
5.40	PutBid
374	StoreBid
0.46	PutComment
45	StoreComment
1.71	AboutMe
6.34	SearchItemsByRegion
5.39	BrowseRegions

Column Families

i3722443462
categories.dummy, *categories.id*, categories.name

i193173044
items.id, *bids.date*, *bids.id*, *users.id*, users.nickname, bids.qty, bids.bid

i1888493477
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3264766123
items.id, *categories.id*

i1175133162
categories.id, *items.end_date*, *items.id*

i2906147889
users.id, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date

i3157175159
users.id, *comments.id*, comments.rating, comments.date, comments.comment

i3563903410
items.id, *bids.bid*, *bids.id*, bids.qty, bids.date

i3128537325
comments.id, *users.id*

i2578518014
items.id, *buynow.id*, *buynow.date*, *users.id*

i2653317939
users.id, *buynow.date*, *buynow.id*, *items.id*, buynow.qty, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i578710568
items.id, *users.id*

i2337785568
users.id, *items.id*, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

i3553045793
users.id, *items.end_date*, *bids.id*, *items.id*

i2817567804
regions.id, *categories.id*, *items.id*, *users.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids, items.end_date

i1683742356
users.id, *regions.id*

i1912786220
items.id, *users.id*, *regions.id*

i590232953
regions.dummy, *regions.id*, regions.name

Plans

BrowseCategories
Get **i2906147889**
Get **i3722443462**

ViewBidHistory
Get **i1888493477**
Get **i193173044**

ViewItem
Get **i1888493477**
Get **i3563903410**

SearchItemsByCategory
Get **i1175133162**, Get **i1888493477**

ViewUserInfo
Get **i2906147889**
Get **i3157175159**

BuyNow
Get **i2906147889**
Get **i1888493477**

StoreBuyNow
Get **i1888493477**
Insert into **i1888493477**
Get **i3264766123**, Delete from **i1175133162**, Insert into **i1175133162**
Get **i2578518014**, Insert into **i2653317939**
Get **i578710568**, Insert into **i2337785568**
Get **i193173044**, Delete from **i3553045793**, Insert into **i3553045793**
Get **i1912786220**, Get **i3264766123**, Insert into **i2817567804**
Insert into **i2578518014**
Get **i1888493477**, Insert into **i2653317939**

PutBid
Get **i2906147889**
Get **i1888493477**
Get **i3563903410**

StoreBid
Get **i1888493477**
Get **i2906147889**, Insert into **i193173044**
Insert into **i3563903410**
Get **i1888493477**, Insert into **i3553045793**
Insert into **i1888493477**
Get **i2578518014**, Insert into **i2653317939**
Get **i578710568**, Insert into **i2337785568**
Get **i1912786220**, Get **i3264766123**, Insert into **i2817567804**

PutComment
Get **i2906147889**
Get **i1888493477**
Get **i2906147889**

StoreComment
Get **i2906147889**
Insert into **i2906147889**
Insert into **i3157175159**
Insert into **i3128537325**

AboutMe
Get **i2906147889**
Get **i3157175159**
Get **i3128537325**, Get **i2906147889**
Get **i2653317939**
Get **i2337785568**, Filter by items.end_date
Get **i3553045793**, Get **i1888493477**

SearchItemsByRegion
Get **i2817567804**, Filter by items.end_date

BrowseRegions
Get **i590232953**

RegisterItem
Insert into **i1888493477**
Insert into **i3264766123**
Insert into **i1175133162**
Insert into **i578710568**
Insert into **i2337785568**
Get **i1683742356**, Insert into **i2817567804**
Get **i1683742356**, Insert into **i1912786220**

RegisterUser
Insert into **i2906147889**
Insert into **i1683742356**

Appendix B

ESON Proofs

Each of the proofs in the section below consists of a claim about the normalization algorithm in Figure 4.6. The steps are the proof are given according to each step in the algorithm.

B.1 All inference of dependencies is sound

`Expand`

When expanding $\mathbf{F}$ and $\mathbf{I}$ we use axioms presented by Mitchell [99] which are shown to be sound.

`BCNFDecompose`

When performing BCNF decomposition, we add two new inclusion dependencies which state the equivalence of attributes in the decomposed relations. For example, if we decompose $R(A, B, C)$ into $S(A, B)$ and $T(B, C)$ then we would add the inclusion dependencies $S(B) \subseteq T(B)$ and $T(B) \subseteq S(B)$. These inclusion dependencies hold by construction.

We also project any FDs and INDs onto the new tables. Any projected FDs are already contained in $\mathbf{F}'$ as a result of the axiom of reflexivity. Similarly, the projected INDs are already contained in $\mathbf{I}'$ aside from a simple renaming of relations to the new names after decomposition. For example, if we decompose $R(A, B, C)$ into $R'(A, B)$ and $R''(B, C)$ then we consider inclusion dependencies involving R using the attributes A and B and change them to reference R'. These hold since R' contains exactly the same (A, B) tuples

as R. A similar argument holds for inclusion dependencies on R containing the attributes B and C.

`Fold`

When removing attributes, all relevant dependencies are already contained in $\mathbf{F}'$ and $\mathbf{I}^{+'}$ since they are projections of other dependencies. When removing relations, there is no need for any new dependencies and we simply remove any dependencies referencing the removed relation from $\mathbf{I}^{+'}$. Finally, when merging relations we simply rename existing inclusion dependencies to reference the merged relation. If we merge $R(A, B)$ and $S(A, C)$ into $RS(A, B, C)$ then we consider separately inclusion dependencies on R involving A and B and inclusion dependencies on S involving A and C. RS contains exactly the same (A, B) tuples as R so any inclusion dependencies involving R will also hold on RS. A similar argument applies for inclusion dependencies involving S and the attributes A and C.

`BreakCycles`

Our technique for breaking cycles is taken from Mannila and Räihä [88]. Suppose we have a cycle $R_1(X_1) \subseteq R_2(Y_2) \cdots \subseteq R_n(X_n) \subseteq R_1(Y_1)$. R_1 is decomposed into $R'_1(X_1 \bigcup Y_1)$ and $R''_1(Y_1 \setminus (\text{attr}(R_1) \setminus X_1 Y_1))$. Three inclusion dependencies are added. $R'_1(X_1) \subseteq R_2(Y_2)$ which is derived be renaming R to R_1 which is sound since these relations contain the same tuples when X_1 is projected. The same argument holds for the inclusion dependency $R_n(X_n) \subseteq R''_1(Y_1)$. Finally, the transformation also adds the inclusion dependency $R'_1(Y_1) \subseteq R''_1(Y_1)$. Since R'_1 and R''_1 both contain values for Y_1 decomposed from R, this dependency also holds. The original authors also show that the transformation is information-preserving.

B.2 All transformations are lossless-join

`BCNFDecompose`

We use a known algorithm for lossless-join BCNF decomposition.

`Fold`

When `Fold` removes an attribute B from a relation $R(A, B)$ it is because there exists a relation $S(C, D)$ with an inclusion dependency $R(A, B) \subseteq S(C, D)$. In this case, we would add the relation $R'(A)$ to the schema and remove R. Note that R can be reconstructed by joining R with S on $A = C$ and projecting A and D (renamed to B), that is $\rho_{B/D}(\Pi_{A,D}(R \bowtie S))$.

If a relation R is removed by `Fold`, it is because there is a bidirectional dependency with a relation S indicating that there is a one-to-one mapping between records in R and records in S. Therefore, we can remove R since it can be recovered by a simple projection of S.

Finally, `Fold` can merge two relations with a common key. If we merge $R(A, B)$ and $S(A, C)$ to form $T(A, B, C)$ we can recover R and S by simply projecting the appropriate fields from T.

`BreakCycles`

When breaking an inclusion dependency cycle, each of the new relations contains a key of the decomposed relation. Therefore, we can perform a natural join on this key to produce the original relation.

B.3 Inclusion dependencies in the final schema are key-based

`Expand`

Two inference rules can generate new INDs. The first is implication via transitivity. If we have functional dependencies $R(X) \subseteq S(Y)$ and $S(Y) \subseteq T(Z)$ then we can infer $R(X) \subseteq T(Z)$. Since we assume existing INDs are superkey-based, Z must be a superkey of T and the new IND introduces no violations. We can also infer new INDs by exploiting the set of functional and inclusion dependencies together. Suppose we have the INDs $R(X_1) \subseteq S(Y_1)$ and $R(X_2) \subseteq S(Y_2)$ as well as the functional dependency $R : X_1 \to X_2$. Then we can infer the IND $R(X_1X_2) \subseteq S(Y_1Y_2)$. Since Y_1 and Y_2 are both superkeys of Y, this new IND is also superkey-based. The second is the collection rule. Since the right-hand side of INDs created by the collection rule is a superset of an existing IND, the new IND is also superkey-based.

`BCNFDecompose`

Assume we have a relation $R(XYZ)$ where X is the key of R and also the functional dependency $R : Y \to Z$. We would then decompose R into $R'(XY)$ and $R''(YZ)$. Suppose we had another relation S with an IND $S(A) \subseteq R(X)$. After decomposing R we would create the IND $S(A) \subseteq R'(X)$ which by construction is also superkey-based since X is the key of R'. The situation is slightly more complicated if we had an IND $S(UV) \subseteq R(XZ)$. After decomposition X and Z are in separate relations. We then have the INDs $S(U) \subseteq$

$R'(X)$ and $S(V) \subseteq R''(Z)$. However, Z is not a superkey of R''. When this situation occurs, we drop the inclusion dependency $S(V) \subseteq R''(Z)$. Note that these inclusion dependencies are not necessary to satisfy IDNF but they are useful to identify possible foreign keys in the final schema.

`Fold`

`Fold` converts INDs which are superkey-based into key-based INDs. Assume we have an IND $R(AB) \subseteq S(CD)$ where C is the key of S. Then this IND is not key-based. However, since C is a key of S, we must have the functional dependency $C \to D$. Therefore, the `Fold` algorithm will identify the attribute B as redundant and remove it from R. This changes the inclusion dependency to $R(A) \subseteq S(C)$ which is now key-based.

`BreakCycles`

Suppose we have a cycle of the form $R_1(X_1) \subseteq R_2(Y_2) \cdots \subseteq R_n(X_n) \subseteq R_1(Y_1)$. As discussed in 4.4.1, breaking this circularity will result in new relations R'_1 and R''_1. There are also new INDs $R'_1(X_1) \subseteq R_2(Y_2)$, $R'_1(Y_1) \subseteq R''_1(Y_1)$, and $R_n(X_n) \subseteq R''_1(Y_1)$. Given that the given dependencies (resulting from the `Fold` step) are key-based, we note that Y_2 is a key of R_2. In addition, since we construct R'_1 and R''_1 such that Y_1 is a key, all new INDs are also key-based.

Appendix C

ESON RUBiS Example Input

Physical relations used as input for both examples are given here. Relation names are shown in bold an attributes which are keys are underlined.

Schema One

i154863668(categories_id, items_end_date, items_id, regions_id, users_id, items_initial_price, items_max_bid, items_name, items_nb_of_bids)
i1888493477(items_id, items_buy_now, items_description, items_end_date, items_initial_price, items_max_bid, items_name, items_nb_of_bids, items_quantity, items_reserve_price, items_start_date)
i193173044(bids_date, bids_id, items_id, users_id, bids_bid, bids_qty, users_nickname)
i2906147889(users_id, users_balance, users_creation_date, users_email, users_firstname, users_lastname, users_nickname, users_password, users_rating)
i3157175159(comments_id, users_id, comments_comment, comments_date, comments_rating)
i3220017915(bids_id, items_id, bids_bid, bids_date, bids_qty)
i3722443462(categories_id, categories_name)
i546546186(categories_id, items_end_date, items_id, items_initial_price, items_max_bid, items_name, items_nb_of_bids)
i590232953(regions_id, regions_name)

Schema Two

i1177375268(buynow_id, buynow_date, items_id)
i1557291277(users_id, comments_id, comments_comment, comments_date, comments_rating)
i1879743023(comments_id, users_id, users_nickname)
i2049737091(items_id, bids_id, users_id, items_end_date)
i2087519603(items_id, categories_id)
i210798434(items_id, bids_date, bids_id, users_id, bids_bid, bids_qty, users_nickname)
i2269844981(items_id, users_id, items_end_date)
i2366332486(users_id, buynow_date, buynow_id, buynow_qty)
i2594090645(items_id, items_buy_now, items_description, items_end_date, items_initial_price, items_max_bid, items_name, items_nb_of_bids, items_quantity, items_reserve_price, items_start_date)
i262196338(items_id, bids_bid, bids_id, bids_date, bids_qty)
i3050485475(categories_id, categories_name)
i3116489164(users_id, users_balance, users_creation_date, users_email, users_firstname, users_lastname, users_nickname, users_password, users_rating)
i409321726(users_id, items_end_date, bids_id, items_id)
i920605840(categories_id, items_id, items_end_date, items_initial_price, items_max_bid, items_name, items_nb_of_bids)
i941409494(users_id, items_end_date, items_id)