A Cost Model for a Fingered Join Operator in Relational Query Plans

by

Vishnu Prathish

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

© Vishnu Prathish 2015

I hereby declare that I am the sole author of this 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.

Abstract

We introduce the finger aware cursor operator for relational join queries. It scans a list of tuples in a finger enabled manner when a nested loop join operation is performed. Using this scan operation, we improve the performance of nested loop join when compared to when compared to conventional scan. To quantify the improvement in performance using fingered scan, a statistic named runs that quantifies the degree of randomness in a list of records is introduced. This statistic is vital in assessing the performance improvement achieved using fingered scan. Using runs statistic as a key ingredient, we develop a cost model that can assign a cost value to the join operation based on underlying fingered scan. We then develop a cost formula and evaluate the cost model against a simulated data set. We show that conventional System R cost model is not sufficient to capture the performance improvement. We then evaluate using the new cost formula and show that it predicts the cost of join operation correctly.

Acknowledgements

First and foremost, I thank my professors, prof. Grant Weddell and prof. David Toman for generous counsel and guidance in every step of the way. This would not have been possible without you. And my readers prof. M. Tamer Ozsu and prof. Richard Trefler for their invaluable contributions. Also,

- Thank you Smrithi, for tolerating me. I promise to be a better husband from now on.
- Thank you Shinzy, for lending a no-questions-asked-helping-hand whenever in need. And thanks CT, for making my life here livable.
- Thank you Achu for being the best possible sister there ever is. Thank you Amma and Achan for making me what I am today and supporting me through and through.
- Thank you Nabo, Cyril, Noel and Clint for the decision we made during that coffee break, which drastically changed our lives.

Dedication

To Smrithi and Swathi.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Contributions Summary 4

1.2 Thesis Organization 5

2 Review 6

2.1 The Basic Join algorithms 6

2.1.1 Nested Loop Join 6

2.1.2 Sort Merge Join 7

2.1.3 Hash Join 7

2.2 Review of System R cost model 8

2.3 Understanding the Cost of Join 9

2.3.1 Cost of nested loop 9

2.3.2 Cost of merge 10

3 Finger Aware Cursors and the Runs Statistic 13

3.1 Runs in a List of Tuples 13

3.2 Introduction to Finger Aware Cursor 16

3.2.1 Representation of Finger 18
3.3 Finger Enabled Join 19
3.3.1 Prerequisites for a Finger Enabled Join 19
3.3.2 Working of Finger Enabled Join 20

4 Estimation of Cost for Finger Aware Joins 23
4.1 Introduction 23
4.2 A Generic Cost Formula 23
4.2.1 About Special Cases 29
4.2.2 Cost of join on Two In-Memory Lists 30
4.2.3 Cost of Finger Enabled Join in a Disk and Memory Model 31
4.3 Calculating Other Statistics in the Join Node 35

5 Evaluation 38
5.1 Introduction 38
5.2 Dataset Generation 39
5.3 Evaluation Overview 40
5.4 Experiments and Results 41
5.4.1 Experiment Overview 41
5.4.2 Verifying and Explaining Anomalies 43

6 Inferences and Future work 45
6.0.3 What's not Covered? 45
6.0.4 Knowing Other Information 46

Appendices 47

A Derivation of Probability of One Interval Falling in Another 48

References 50

List of Tables

1.1 Statistics used by System R Cost Model 2

2.1 Terminology Used in this Thesis 12

4.1 Statistics Available for the Optimizer for Files I and O 25

4.2 Cost Components in the Formula 29

4.3 Statistics Available for the Optimizer for Fingered Page Scan 34

4.4 Selectivity Estimates of Various Operators 35

4.5 Runs and Size Estimates of Various Operators 36

List of Figures

1.1 System R Architecture 3

3.1 List of integers with runs 14
3.2 Runs distribution in the list of integers 14
3.3 A Representation of Finger Aware Cursor on a list of Integers 17
3.4 Simple Join using Fingered Scan on two lists of Integers 20

4.1 Fingered Join on a Disk and Memory Model 34

5.1 Experimental(Nested loop) vs Predicted(System R) values 42
5.2 Experimental(Fingered Scan) vs Predicted(System R) values 42
5.3 Experimental(Fingered Scan) vs Predicted(New cost model) values 43

Chapter 1

Introduction

Query optimization is a prominent issue in relational database management systems(DBMS) with over forty years of research behind it [6, 1]. Join and join like operators, being the most expensive of relational algebra operators to optimize, has been at the forefront of this research. The renowned System R [3] paper, the pioneer in the field, laid the foundation of query optimization and presented one of the first notion of cost based query plan selection.

System R paper considers Nested Loop and Sort Merge as two major join methods and build a way to estimate the cost for these two methods. Over the years, various join algorithms have evolved and several cost optimization techniques have been proposed adding to the original System R cost model. As the database research currently stands, the most common join algorithms used in the industry are variants of Hash, Nested loop and Sort merge joins. It would be oversimplification to classify the algorithms into these three. However, it is safe to say that these three and their various derivatives are widely used in most DBMS engines in industry. While the relevancy of some of them have reduced and some of them have adapted to changed requirements of modern network and disk/memory access speeds, there is at least one scenario in which one outperforms others.

The objective of this thesis is to build a cost model for an alternate approach which brings out the features of both sort merge and nested loop algorithms by measuring the degree of randomness in data forehand. First, we introduce the Fingered Scan Operation which is a new method of scanning records from the disk or memory. The scan occurs through two APIs exposed by the operator, getNext() and getFirst(). Similar to a database cursor, it returns the next record and first record in the list respectively. The additional and essential feature of this scan is that it remembers the value of last scanned record. This allows it to compute the *change in ordering* from the last scanned record to the current

Table 1.1: Statistics used by System R Cost Model

Statistic	Description
NCAD(T)	The cardinality of relation T
TCARD(T)	The number of pages in the segment that holds tuples of relation T
P(T)	The fraction of data pages in the segment that hold tuples of relation T
ICARD(I)	Number of distinct keys in the index I
NINDX(I)	Number of pages in Index I

record. The operator makes improved decisions on where to move the scan pointer based on this change in ordering.

This fingered scan operation results in a performance improvement for a join operation. This improvement can be attributed to the reduction in the number of records that need to be scanned because of the improved decisions made by the operator. But the list of tuples involved in the join operation needs to confirm to a set of constraints for the finger enabled scan to attain this performance improvement. The main constraint is that it required a completely sorted inner relation; like a join on the key column of a table in relational database. This improvement, which results in a reduced cost for join, can be either captured in terms of the reduced time required to complete the operation or in terms of the reduced overall number of records read in the process. In this thesis, the second method is selected to highlight the performance improvement achieved.

Once the improvement in performance is shown, we need to quantify this improvement. This is necessary because existing System R style cost model doesn't have enough information about ordering in data to capture the performance improvement. The first step in assessing the new cost is collecting all the required statistics about the data. In addition to the statistics used by System R, 1.1 we also need additional information that can be used as a metric to measure the decisions made by the fingered scan operator in response to change in ordering. These decisions depend on the degree of ordering in the list of records being scanned. An effective way of measuring the degree of ordering is by using a statistic called runs. Previously introduced by Bradley et. al [2], runs is a distribution free statistic which measure the total number of subset of observations in a list of observations where each of such subset is totally ordered.

The runs statistic, along with the other statistics used by system R, are sufficient to build up a cost model to predict the cost of finger enabled scan accurately. Using this additional information, we build a cost formula which calculates the total cost of finger enabled join. In addition, we also calculate runs for result sets of other operators like

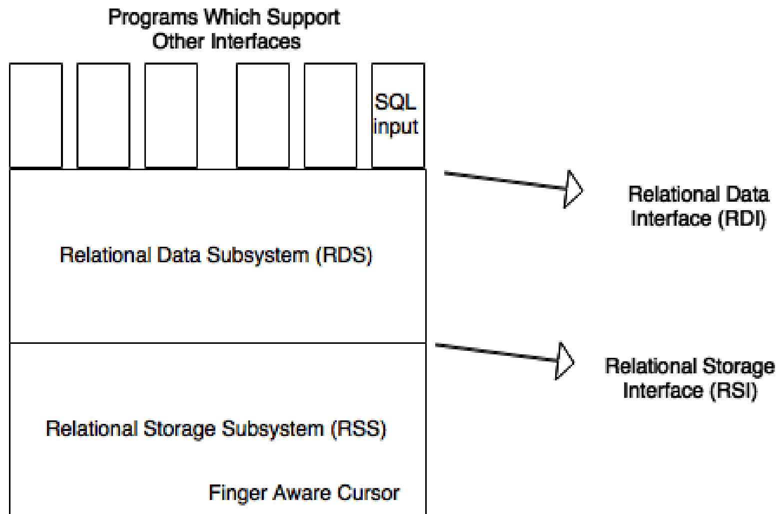

Figure 1.1: System R Architecture

union, sort, subtraction etc taking the number of runs in the origination relations as input.

Once the cost model is built, we evaluate it using simulated data set containing runs and show that it achieves the predicted performance. A data set generator engine generates two files, outer and inner, with a given size, runs and key attributes. An implementation of finger enabled join algorithm takes these two files as input and produces a cost output along with results. An implementation of conventional nested loop join also does the same in an identical environment. The result sets are compared for accuracy before comparing the costs. The cost predicted by System R cost model is plotted against the cost output by nested loop join to show that they conforms. Then, the System R cost is plotted against cost of finger enabled join to show that there is a large gap between predicted cost and actual cost. Then, we plot the cost of finger enabled join with the predicted cost with the new cost formula to show that the gap is covered and the new cost formula can predict the performance improvement well.

System R has two subsystems, a storage subsystem named Relational Storage Subsyste (RSS) and a data subsystem named Relational Data Subsystem (RDS) as mentioned in table 1. As quoted in the original paper [1], storage subsystem handles *devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, backout, transaction recovery, and system recovery.* RDI provides *authorization, integrity enforcement, and support for alternative views of data.* Fingered scan belongs to the storage

subsystem which reads the files from disk and returns tuple by tuple.

1.1 Contributions Summary

Major contributions of this work are:

- Introduce join operation with underlying fingered scan as an alternative for existing access paths.
- Provide a generic cost model for finger join that enables the optimizer to evaluate the cost against other query plans.
- Apply the generic cost formulae for two real world situations. One, a linked list of records in memory. Second, a file stored in disk loaded into memory as pages.
- Evaluate the approach with a generated data set to see that cost formula is effective. Here we,
  - Show that fingered scan improves cost of join operation over nested loop join
  - Show that System R cost model is inadequate for capturing this improvement attained using fingered scan.
  - Show that the new cost model introduced in this thesis adequately captures this improvement

1.2 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we lay out the basic background concepts required to appreciate this work along with some of the previous work done in the area. This covers a preview of System R cost model relevant to this work. This includes a quick coverage of relevant join algorithms and use cases. We also understand the notion of cost when applied to join operator and look at the cost of relevant join algorithms. In Chapter 3, we explain the notion of fingered scan and runs statistic that can capture the randomness in data. We also explain some necessary conditions required in the data set so that the subject handled in this thesis is applicable to the data. In Chapter 4, we introduce the cost formula built assuming that a join operation is performed with underlying fingered scan in play. We then apply the formula for a single level store as well as a disk and memory model. In Chapter 5, we first evaluate the model prerequisites explaining the simulator, data generator and experiment overview. Then we analyze the results to validate the cost formula. In chapter 6, we look at some inferences and future work.

Chapter 2

Review

2.1 The Basic Join algorithms

Though there are several modifications applied on them on various contexts, the core three join algorithms are,

- Nested Loop Join
- Sort Merge Join
- Hash Join

2.1.1 Nested Loop Join

The oldest yet widely popular join algorithm works on the idea that every record in the outer file needs to be scanned for every record in the inner file. For every such pair, the qualifying records are picked up based on the Join predicate.

```
foreach record r_i in Outer R do
    foreach record s_i in Inner S do
        if r_i == s_i then
            add < r_i, s_i > to result;
        end
    end
end
```

Algorithm 1: Pseudo code for simple nested loop join

This simple algorithm considerably sped up using a block based nested loop, where instead of records, a buffer consisting of many pages containing the records of outer are loaded and compared at once with the records in a page of inner relation.

```
foreach page po in block Bo in Outer R do
    foreach page pi in Inner S do
        foreach record r_i in po do
            foreach record s_i in pi do
                if r_i == s_i then
                    add < r_i, s_i > to result;
                end
            end
        end
    end
end
```

Algorithm 2: Pseudo code for block nested loop join

2.1.2 Sort Merge Join

In sort merge join, both inner and outer relation is sorted and then a merge is performed to match the join predicate hence picking up the qualifying records. It is widely used when the records are already sorted or only a minimal effort is needed to be spent for sorting purposes.

The gain in using sort merge over nested loop is in situations where the effort needs to be spent on sorting is minimal. In such cases, both the files are only scanned once. The component that affects the cost is the effort spent on sorting [7] and therefore, this component often becomes the decisive factor.

2.1.3 Hash Join

While the actual working of hash join vary depending on the size of inputs, the general idea is that one of the relations are converted into a hash table with the join attribute as key. And then for every record in the outer table, a hash look up is performed to match the join predicate. Due to its nature, hash based algorithm is typically used for equi-joins.

2.2 Review of System R cost model

System R was the seminal work in the field of relational queries and optimization. It introduces a subsystem, a cost model and proves that relational queries can work with guarantees in a real world system. It built up the theory that most modern DBMS engines still use today. Since our cost model can be viewed as an additional component on top of System R cost model, we give a preview of System R cost model focusing on sections that are relevant to this thesis.

Sellenger [12] introduced the very first cost model for the IBM system R in 1976. She introduced the concept of calculating the cost of a query based on disk reads and CPU time within the storage subsystem used by System R.

This storage subsystem uses RSS scan, a disk scan which returns tuples one by one from the disk. The scan supports OPEN, NEXT and CLOSE calls, primarily meant to read the relation sequentially. The NEXT operation is similar to the getNext() operation when using fingers. There are two ways of doing the scan, a segment scan which reads all the pages occupied by the relation and an index scan which reads through the leaves of the B-tree style index getting the tuple references. In the scope of this thesis, we are not considering an index scan. Segment scan is basically reading pages one after other. During the evaluation phase of this work, we adapt segment scan to a record level reading records one by one from disk.

System R uses search arguments(SARGS) to filter an index scan before returning to predicates. The selectivity of these search arguments predicts the approximate amount of tuples returned after the filter process as a function of total size of relation. The cost of scan for a single relation is determined as a weighted measure of number of pages read and CPU calls. Once the cost of scanning a single relation is determined, the cost of a nested loop join can be determined using the formula,

$$C_{NESTED-LOOP-JOIN}(path1, path2) = C_{OUTER}(path1) + N \times C_{INNER}(path2) \quad (2.1)$$

where C_{OUTER} and C_{INNER} refers to the cost of scanning individual relation considering the predicates.

The cost equation used in the evaluation of this thesis is an adaptation of equation 2.1. Even though we haven't considered an index scan, it is necessary to predict the estimated size of the relation and other parameters that would be the outcome of an operation. Such an estimate would help optimizer to make decisions on the various operator nodes in the query plan. Hence, we predict a list of estimates for relevant statistics based on a single relation scan.

System R deals with more literature than that is have covered in this review. It talks about cost of sort merge join, dynamic programming to do the join ordering, cost of nested queries and more. But for the scope of this work, the literature we have covered on System R is sufficient.

2.3 Understanding the Cost of Join

Every Join algorithm perform better than others in at least one scenario. Nested loop is picked when relation is quite small, sort merge is picked when cost of sorting is minimal and hash join is picked when the type is equi-join with a hash table that fits in the memory. Optimizer picks the right one that can minimize the I/O operations in a particular scenario. In system R, this is done by first building up a search space with all possible query plans. Then cost for each of the plan is calculated by summing up the costs for individual operators involved in the query plan. The one with net lowest cost is picked. Hence, the cost is traditionally counted as a function of number of pages accessed from disk and CPU time. Let's discuss a simple nested loop Join without paged or block access to understand the idea.

2.3.1 Cost of nested loop

The cost of naive nested loop, as mentioned in Algorithm 1 is relatively straight forward. The cost as a rough estimate of complexity is the size of outer relation multiplied by the size of inner relation. We will build up from a simple in memory nested loop.

Cost of In Memory Nested Loop

This algorithm assumes that the basic unit of cost is accessing each record of the list from memory. Outer relation is read once and inner is read once per one record in outer relation. Hence,

$$Cost = Size(O) + Size(O) * Size(I) \tag{2.2}$$

Here, cost is measured in terms of number of records read. Many existing systems ignore this CPU cost as we are more concerned about the cost when data becomes larger and doesn't fit in memory. In such a case, the cost of I/O from disk far outweighs this cost

of CPU cycles.

Cost of Disk Based Nested Loop

Blocked nested loop, as mentioned in Algorithm 2, is an improvement over conventional nested loop in which files can be read as blocks containing groups of records. And nested loop join is performed per block instead of per record. The differences in cost between reading from disk and reading from memory means that the cost mentioned in 2.2 is no longer relevant. The blocks containing outer relation is read once and blocks containing inner relation is read once per number of blocks in outer.

$$Cost = B(O) + B(O) * B(I) \tag{2.3}$$

This is the cost measured in terms of blocks. The block size is usually determined by the available memory as calculated by optimizer. Since this is done dynamically, the statistic optimizer keep is the number of pages occupied by both O and I. Number of blocks per file is then calculated by,

$$B(R) = \frac{Pages(R)}{Blocksize} \tag{2.4}$$

For the purposes of this thesis, we take pages as the basic unit of access and do not dive into statistics associated with blocked access.

2.3.2 Cost of merge

The worst case cost and complexity of sort merge join is same as nested loop. But there are a couple of cases where the optimizer doesn't need to resort to this high cost. If the sort is performed either somewhere else down in the query plan, we get a sorted relation. It may be the case that the relation is already sorted as well. In these cases, we are able to save upon this cost and the overall cost then becomes the cost of merge alone. Which is,

$$Cost = Size(R) + Size(S) \tag{2.5}$$

which is negligibly low compared to sorting.

Query optimization as a research topic is one of the oldest topics in Computer Science research. Generic cost models for SQL queries have been extensively researched since 1980's. Codd [4] laid down the relational basis for database query languages. System R paper introduced the first cost model based on access paths [12]. Join ordering still works on the basic algorithm devised by System R paper [3] [1]. Since then, several other works [13] [9] [10] talk about statistic based query optimization and the different kinds of statistics to be used for the purposes.

In this thesis we use a new such statistic called runs. Runs as a statistic to detect the degree of randomness was introduced even earlier by Bradley [2]. While runs test has been used in several fields including quality control [15] and detecting virus infections [15], an attempt to use runs as a statistic in query optimization engines is novel. Our work focuses on using runs to detect the degree of ordering and use the measure to use a finger based join operation.

Using fingers for faster look ups was first introduced in by Guibas et. el. [5]. Even though it serves a similar purpose, fingers in this thesis are used in a different context with different fields and different methods operating over it. We have getNext() and getFirst() calls operating over a finger. getNext() is very similar to *FETCH next record* operation on a database cursor [14] and getFirst() is similar to resetting the cursor to beginning.

We build upon all the aforementioned past works, however, the core theme of this thesis is unique. But our work may only be a step towards this direction. There are several inferences that could directly relate to this work. We give a brief look at these future works in the inferences section.

Name	Explanation
field/column	A single item of information, such as email id or employee name.
record	A single implicitly structured data item composed of fields.
file	A regular disk based file containing a series of records that exposes APIs to read, write and seek the records
List (L)	A list of records. Used in a context where a disk based file is not relevant. Such as an in memory list of records
Column (N)	The list of values of Nth field in all records in the file
r, s	variables used to represent individual records
O	Outer file. The file used in the outer loop of a typical nested loop join or finger enabled join
I	Inner file. The file used in the inner loop of a typical nested loop join or finger enabled join
R	A generic file that could be O or I containing a series of records
Size(R)	The total number of records in file R.
Pages(R)	Total number of pages occupied by R on disk.
B(R)	Cost of reading a block of file R
$Run_i(R_N)$	A list of observations read from N^{th} column, i^{th} run.
$Page_i(R)$	i^{th} page occupied by file R

Table 2.1: Terminology Used in this Thesis

It can be observed from the review that sort merge join would be the obvious choice, provided every relation is already sorted. But unfortunately, in reality, the query optimizer would have to choose either sorting and merge approach or to choose one of the other join algorithms. Typically, query optimizers choose sort merge join when the resultant algorithm achieves near linear run time, which means that the effort that needs to be spent in sorting is minimal. And Nested loop when there is a larger effort involved in sorting. In the next chapter we introduce finger aware cursors which can remove the process of *choosing* altogether, when only one of the relations involved in the process is sorted on the join field.

Chapter 3

Finger Aware Cursors and the Runs Statistic

In this section, we define the concept of *runs statistic* and examine how a *finger aware cursor* would work in real scenario.

3.1 Runs in a List of Tuples

Runs Assertion

(R, L, A, B, N, Runs(R)): For a linked list L of instances of record type R, and columns $A\&B \in R$ encoding intervals ($A \leq B$ and $C \leq D$), there are $Runs(R, A, B)$ runs.

Explanation

We will build up to a generic record type with two fields A, B and a generic runs definition from a simpler case of a linked list of positive integers. Such a list is sufficient to explain the two key ideas we are considering in this section, namely, the finger aware cursor and runs statistic.

Consider the array of positive integers given in Figure 3.1. This list not sorted in the classical sense of total ordering. Figure 3.1 shows the same array of positive integers in a graph plotted against their index in the array on y-axis. From the graph, you can see that elements with indexes 0 to index 4 are sorted. So are the indexes 5 - 10, 11-15 and

33	45	66	68	72	45	67	69	78	79	84	67	69	88	90	99	12	29	35

Figure 3.1: List of integers with runs

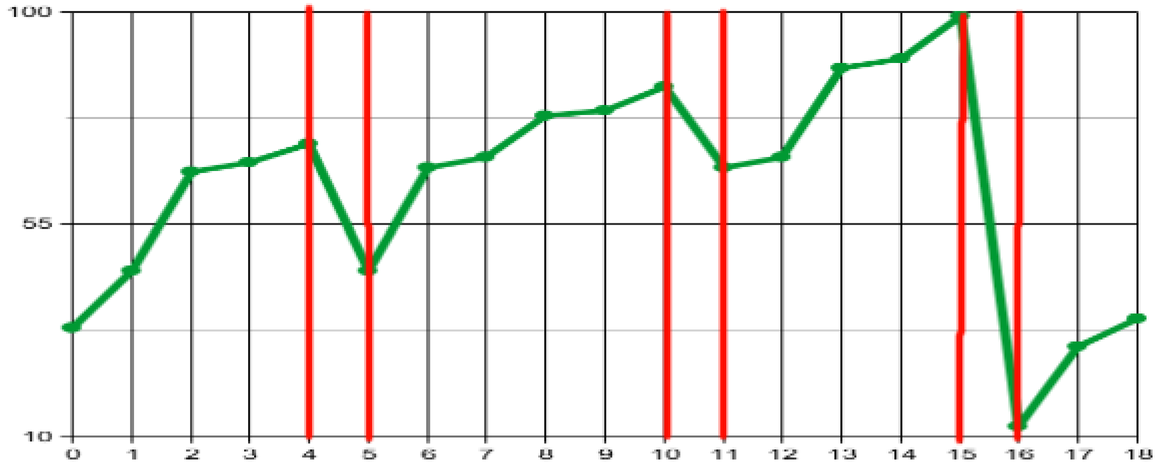

Figure 3.2: Runs distribution in the list of integers

16-18. Even though the complete array is not sorted, there are ordered units within the array. The areas marks in red are the points where a change in the ordering occurs in an otherwise regularly sorted-ascending array of integers.

This array can be treated as a list of values from an *employee_id* field from an employee database which is read sequentially using a cursor. An attempt is made to capture the degree of ordering in such a list of integers (records in a broader sense) where breaks are often observed in the ordering. *Runs*, is a good measure of this degree of ordering. In this context, our definition of runs is very similar to what is defined by Bradley [2] in 1968. He defines a single run as,

An unbroken sequence of increasing or decreasing observations.

The total number of runs in Figure 3.1 is 4. It is calculated by adding one to total number of changes in the ordering in the list. Number of runs in list of integers is an effective way to capture the partial ordering in data. In order to fulfill the requirements of the work we cover in this thesis, a more comprehensive definition of runs based on a two field record is required. Such a definition of Runs() is given in Chapter 4.

Programmatically, number of runs for a list of positive integers is defined and collected

like this,

foreach *Integer r in R* **do**
  if $r_i < r_{i+1}$ **then**
    runs++;
  end
end

The characteristics observed on a list of integers here can be extended to a list of records with column / field names. The notation $Runs(R, N)$ is used to denote number of runs in field N if the list of records are scanned successively from the disk in the order of storage. Table 3.1 shows the various cases and explanations for the values assumed by $Runs(R, N)$.

$Runs(R, N)$	Explanation
1	List of records is fully ordered with respect to column N.
Size(R)	List of records is reversely ordered with respect to column N.
k	Indicates the degree of partial sorting among the records. Higher the k, lower the average length of individual sorted chunks

Consider a record $r \in L$ with structure as shown in Structure 3.1, and that a scan operation takes place on this list of records on *employee_id* predicate.

```
struct record
{
    int employee_id;    //N=0
    int salary;         //N=1
    struct record *next;
}
```

When the records containing *employee_id* gets scanned one by one, the order in which this scanning takes place determines the number of runs for that field. The value for $Runs(R, employee_id)$ is pre-calculated as a statistic for field *employee_id*. Similarly, $Runs(R, salary)$ is calculated for salary.

By this definition, number of runs is the number of times a partially ordered series of records go out of order. Runs() statistic along with the size of records are the two parameters optimizer base its decision regarding the choice of a finger based join or other types.

$Runs(R, employee_id)$ forms an important component of the cost formula. It works as a statistic to incorporate the advantage incurred by fingered scan. DBMS engines already

collects statistics such as histogram [8] which essentially gives meaningful information about data distribution. Currently, they are used for the purposes of selectivity estimation [11], approximating query results [10] and several others. Runs() is a distribution free that talks about the degree of randomness in data.

Runs Statistic with Two fields

For the purpose of building a generic model which can capture more than a list of records or integers as observed till now, we use a different definition of runs. Two fields in the record contribute to a singles runs statistic as opposed to one field we considered till now. Consider a record $r \in R$ with structure shown below.

```
struct record
{
    int A;
    int B;
    struct record *next;
}
```

Our definition of runs on a set of records changes from what we have seen till now. Runs statistic is now denoted by $Runs(L, A, B)$ and computed slightly differently as explained in Section 4. The two fields A and B are representative fields used in a join operation with both fields contributing to the definition of run. There are special cases attached to them. For instance, in first case, it could be database fields like employee id or salary. In another one, each record would stand for statistics on a page where A and B could be the min() and max() of the page for a particular record. The latter is used to generalize the cost formula in this thesis for a two level store. And in the case of former, the formula gets converted to the one we have seen in previous section.

3.2 Introduction to Finger Aware Cursor

The relatively straightforward idea behind fingered scan can be quickly summarized for two lists of positive integers,

In a join operation on two linked lists of integers O and I, by keeping a pointer to the currently read item in O, we can avoid the need for full scan of I if I is completely ordered. If I is completely ordered, a merge can be performed between every run in O and entire I to achieve same result set as a complete nested loop join.

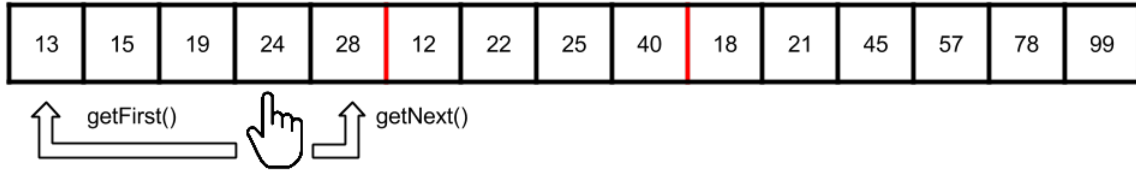

Figure 3.3: A Representation of Finger Aware Cursor on a list of Integers

An inevitable condition is that I has to be completely ordered. If O contains 4 runs and I is fully ordered (1 run), four merges, one each with every run of O need to be performed with I. To do this, two pointers, one each in O and I that scans through the lists are required. The pointer in O also needs to trigger a reset of pointer in I at the end of every run in O.

An simple finger with getNext() and getFirst() attached to it is given in figure 3.3. A finger is basically an extension to the concept of these two pointers. They work on records instead of integer lists. And the scan operation could take place in disk or memory. It scans, remembers the records and facilitates the join operation using Runs(O) merges between O and I. To explain this better, we will see possible representations of a finger enabled scan in two different scenarios,

Simple Finger Aware Cursor

At its simplest form, a finger aware cursor is simply a pointer which keeps track of the currently seen record in a linked list of records. Two operations work on them,

1. getFirst() - Gives the very first record in the list and resets the finger to beginning.
2. getNext() - Gives the next record in the list and moves the finger to the next record.

This is an in memory linked list that completely fits in memory from the first to last record. In this case, it is analogous to a database cursor. A database cursor is a mechanism which allows to perform operations on the result set returned from a query in a row by row manner. It always point to the current row being processed and exposes an API like getNext() which returns the next row in the result set. A finger can be thought of as an extension to this concept which works on list of records. In addition to returning rows only when required, a finger ensures that the row is scanned and computed only when required, providing an additional level of performance improvement.

A Disk and Memory Model

When the concept of finger is further extended to a two-level store with a disk and memory, a linked list that completely resides in memory is no more a suitable term. The new data structure is a file that resides on disk. It contains records that are partially read into memory when required. This requirement more often comes as a db scan operation which reads records in sequence. We assume that DBMS has complete control of number of pages to be read from disk and loaded in to memory whenever required.

In such a file, fingers that work in record level and the ones that work in page level need to be introduced to account for speed difference. This is because the cost of loading a page to memory often far outweighs the cost of in memory getNext() or getFirst() calls. Even though there are two different costs associated with it, the APIs exposed by them namely, getFirst() and getNext() remain same and does not deviate from the functionality mentioned in section 3.2.

For convenience, the finger that operates on pages are referred as a major finger and that operates on records are referred as minor finger. When a list of pages is considered in this manner, the speed difference between major and minor finger in any real scenario will be so large that operational cost of minor finger can be ignored. Such a finger that scans for records will be placed in the leaves of the operator tree where the scan operation is a segment scan on files located in disk.

3.2.1 Representation of Finger

A C-like representation of finger data structure is quite useful to visualize the concept. A list of records declared in C like the following,

```
record* list = malloc(size * sizeof(*record));
```

will have finger pointing to its very first record as a regular C pointer.

```
struct record *finger = list;
```

and two methods getNext() and getFirst() operating over it which returns the next record in the list and the first record in the list respectively.

```
struct record getNext() {
    finger = finger.next;
    return finger;
}
```

```
struct record getNext() {
    finger = list;
    return finger;
}
```

3.3 Finger Enabled Join

3.3.1 Prerequisites for a Finger Enabled Join

The following two prerequisites should be met by two files O and I that are involved in a finger enabled join operation.

1. The records in I should be completely ordered with respect to the corresponding term in join predicate.

2. The number of runs in the O should be reasonable enough to justify the use of fingers.

Th first condition that mandates complete ordering for I, is required to perform the merge between individual run in O and the inner relation. Second condition is not a mandatory prerequisite. Nonetheless, the query optimizer should be able to chose when to use a finger enabled join from the statistics we have available as opposed to another join method. It is possible build enough tools to evaluate the second prerequisite. A cost formula is the first step towards it. However, for the scope of this thesis, we are not dealing with the second condition. We assume that, with any number of runs, finger enabled join is justified.

Such situations are not a rarity. In a typical relational schema, the tables containing a key is always fully ordered which would make up a perfect inner relation. A join on the key as inner relation with any other table satisfies the prerequisite that inner should be completely ordered. The number of runs in a typical column will vary between 1, if fully sorted, and $Size(R)$ if completely unordered. Our tests suggest that any number of runs less than $Size(R)$ is good enough except when it begin to approach the proximity of $Size(R)$. When it does, the overhead associated with fingered scan tends to outweigh its benefits, otherwise, satisfying prerequisite 2.

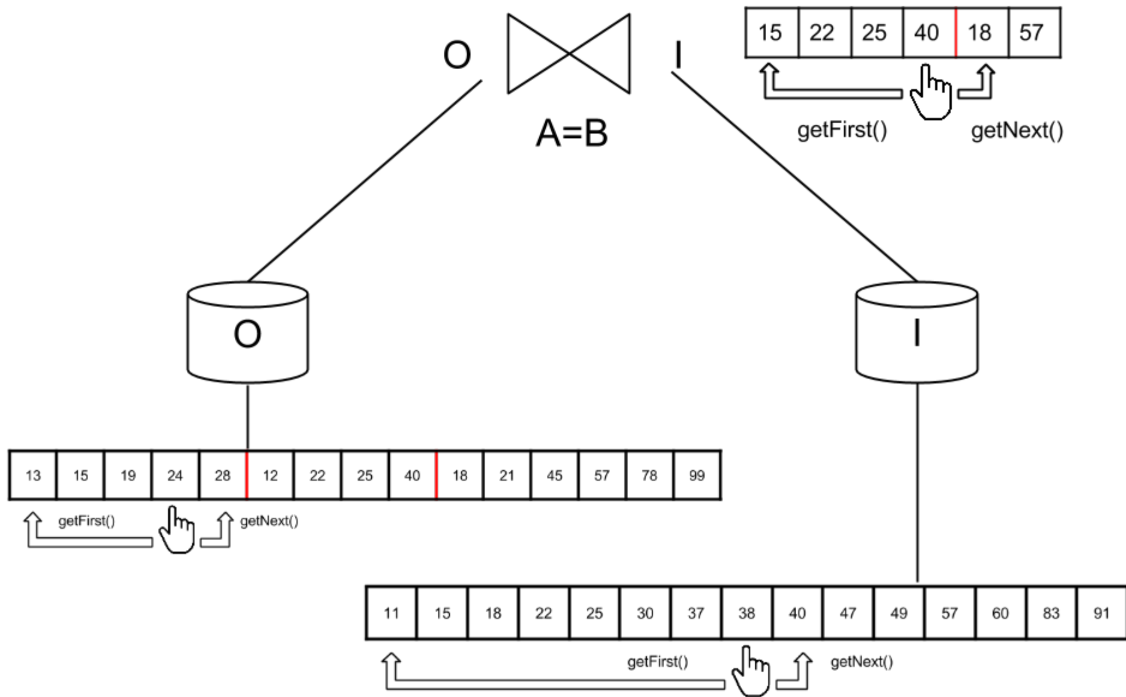

Figure 3.4: Simple Join using Fingered Scan on two lists of Integers

3.3.2 Working of Finger Enabled Join

Consider the initial state where both the fingers are just initialized in a simple join as shown in Figure 3.3.2. The file with a column $Runs(R, A, B) = 1$ is labelled to be the inner (I) while the other one is labelled outer (O). Fingers are reset and rests on the first record.

1. A getFirst() is called on the join node in the operator tree (As indicated in the Figure 3.3.2) to obtain the first join result. The join operator node then issues a getFirst() call on the leaf O and I to obtain the first set of records . At this point,

   - Fingers in both the leaf nodes O & I are reset to first position. Records in that position is read and passed on to the Join operator
   - In the leaf of the operator tree that represents a file (Inner and Outer), the finger reads next record. Finger is incremented and record is passed to the operator.

- In the join operator node, operator checks if the two records in hand matches the join predicate. If so, the results are appended to a result set. It calls getNext() on the leaf which returned lower value of the two. But the old records are not discarded yet.

2. As a result, in a leaf, (O & I) getNext() operation continues until either of these conditions are met

   (a) End of file is reached.

   (b) A getFirst() request is received.

   Once either of the two conditions are met, the finger resets to first record.

3. Steps 1 and 2 are repeated until a run is encountered in the outer file O.

4. Join operator node detects a run using the record that was retained when getNext() was issued. A comparison with this record is made to see if fetching this particular value resulted in encountering a run.

5. Once a run is encountered, the join node issues a getFirst() to the inner file.

6. These steps are repeated until all results are fetched.

As you can see, the change in ordering in the outer file is the only event that triggers a finger reset in the inner file. There is considerable savings in not reading every record each time. Hence it captures the optimization using partial ordering in data effectively.

Duplicate Handling

Apart from the working described above, duplicate values in both O and I need to be handled separately. To do this, we place a second finger in O in the same place as first finger. For all the practical purposes, the second finger moves together with the first. Except when the value being compared is a duplicate. To handle duplicates, we keep it a rule that when the value being compared by the finger in I and the finger in O is same, the finger in I is the one that moves forward. The rules makes it easy to detect duplicates. On occurrence of a duplicate in I, the second finger in O stays in its place and stays as a reference pointer to the beginning of duplicates. The first finger moves ahead as usual. The finger which stays in its place is reset back to the position of first finger on occurrence of a non duplicate value.

Once we encounter a non duplicate value in I, (which is higher than the value finger in O points to as $Runs(I, N) = 1$), we move the finger in O forward. At this point, three cases ensue:

1. A non duplicate, higher value.
2. A duplicate value
3. A lower value, ie, a run.

In case 1, we move both fingers together, considering it as a single finger. If the record being currently pointed to has only one finger (Already encountered duplicates), we reset both the fingers to this position. In case 2, we keep the second finger intact and move only the first finger. In case 3, we reset the finger in I as usual.

Chapter 4

Estimation of Cost for Finger Aware Joins

4.1 Introduction

The cost model introduces a method to calculate cost of the finger enabled join operation. The end result of our cost analysis is formula what calculates a numerical cost value for join using the statistics already available to the optimizer.

We first derive a generic cost formula considering a list of records with certain properties without being concerned about the placement of finger. The properties include two fields A and B and a calculation of runs statistic based on these two fields. This generic formula, once derived can be applied to two possible finger placements as two special cases. First, an in memory linked list of records. Second, a two level disk and memory based system where pages containing the records are first loaded into memory. Once loaded, records are read one by one in-memory. In the second case, we consider the in memory operations as free, allowing us to use the same formula there. Hence, this record-by-record case and page-by-page case, both can be fit into the same formula as two special cases.

4.2 A Generic Cost Formula

To build a generic cost formula, we first define a more general definition of a run as promised in Section 3.1.

Consider a record $r \in L$ with Size(L) records, each record r_i with two fields A and B, denoted by tuple (A, B). This type of record with two fields serves as the record type used in the derivation of generic cost formula.

We first define the interval criteria that is an essential prerequisite for the record r with two fields A and B to work as a generic record type. Then we define runs based on the two fields. We use this information and other statistics shown in 4.1 build the cost formula. The process of building cost formula take place in two sections, one each for the costs incurred by O and I. While cost incurred by O is relatively straightforward to calculate, cost incurred by I has to consider duplicates, boundary corrections, etc.

Interval Criteria for A and B

The fields (A, B) meets the criteria, for every record $r_i \in L$,

$$r_i.B \geq r_i.A \tag{4.1}$$

That is, B succeeds or equals A in terms of ordering and hence (A, B) forms an interval.

Definition of Runs

Now, consider another such record r_{i+1} which has similar fields and follows interval criteria 4.1. A record r_{i+1} is ordered with respect r_i if the pair of records (r_i, r_{i+1}),

$$r_{i+1}.A \geq r_i.B \tag{4.2}$$

That is, the second field of first record should precede the first field of second record. Unlike the definition of run we have encountered till now, a violation to this rule in a list of records is considered to be the end of a run. Hence, a Runs() is defined as the total number of occurrences in a list of tuples in which this condition is violated. That is, for all r $\in$ L the violations of, equation 4.3 is a run.

$$r_{i+1}.A \leq r_i.B \tag{4.3}$$

Into Cost Formula

Consider two files outer and inner with equi-join on columns (A,B) & (C, D) as follows.

$$O \underset{(A,B)=(C,D)}{\overset{\rightarrow}{\bowtie}} I$$

(A, B) and (C, D) are the fields that form an interval. The join condition (A, B) == (C, D) degenerates into A == C in a single column scenario when A == B and C == D. This case is considered as a special case later in this chapter. For now, the file I with Size(I) records ($r_1 ... r_{Size(I)}$) contains these two fields and forms an interval. Additionally, let the property 4.1 be true for I. Hence,

$$Runs(I, C, D) = 1 \tag{4.4}$$

File O with Size(O) records meets the property 4.1 as well. Additionally, the file O also contains a fixed number of runs indicated by $Runs(O, A, B)$. $Runs(O, A, B)$ is an arbitrary number greater than or equal to 1.

Table 4.1 shows all the required statistics available for the optimizer for files I and O.

Name	Symbol
Number of records in file	$Size(I)$, $Size(O)$
Cost of getNext() on file	$CN(I)$, $CN(O)$
Cost of getFirst() on file	$CF(I)$, $CF(O)$
Number of runs in file I for columns (A, B)	$Runs(I, A, B) = 1$
Number of runs in file O for columns (A, B)	$Runs(O, A, B)$
Average runlength of columns (A, B)	$runlength(R, A, B)$
Average runlength of columns (A, B)	$runlength(R, A, B) = Size(I)$
Number of unique values in columns A, B, C, D	$U(R, A)$, $U(R, B)$, $U(R, C)$, $U(R, D)$

Table 4.1: Statistics Available for the Optimizer for Files I and O

Now, we split down the individual cost components of various actions involved in the Fingered Join process.

Cost of Scanning O

Placing the finger in the beginning of O needs a getFirst() which costs,

$$CF(O) \tag{4.5}$$

The outer file is issued a getNext() once for every record which costs,

$$CN(O) \times Size(O). \tag{4.6}$$

This sums up the costs involved for accessing outer file, ie. the sum of 4.5 and 4.6,

$$C_{outer} = CF(O) + CN(O) \times Size(O) \tag{4.7}$$

Cost of Scanning I

Calculating cost for I is not trivial as it involves several back and forth finger movements between records. Placing the finger in the beginning of I needs one getFirst() call, costing,

$$CF(I) \tag{4.8}$$

For every run in O, there is another getFirst() call for I, resulting in the cost component,

$$CF(I) \times Runs(O, A, B) \tag{4.9}$$

However, an important ingredient in the cost formula is the cost for total of getNext() calls issued to I. For every run in the outer, we have to scan the inner up to a certain k^{th} record $r_k \leq Size(I)$ in order to perform the merge. Hence we are looking at cost component,

$$Runs(O, A, B) \times k_{avg} \times CN(I) \tag{4.10}$$

where k_{avg} = average of all such k values in I for every run in O. The value of k_{avg} depends on the distribution of data. But we later consider a reasonable approximation of this value based on a uniform data distribution assumption.

Next significant cost component originates from accounting for duplicates. This component is the result of getNext() calls on I for every duplicate value in O. Refer 3.3.2 for detailed explanation. For a record in O, the number of getNext() calls in I will be equal to the number of consecutive duplicate values in I for that particular record. This component is non-trivial to calculate because we are considering two fields per record. For a join involving only one field, definition of duplicates refers to the conventional definition of duplicates where values being same and being able to be compared with an *equal to* (=) operator. But when considering two fields forming intervals, an interval (A, B) is considered duplicate of another interval (C, D) only if,

$$A \leq C \ \& \ B \geq D \text{ or } A \leq C \ \& \ B \geq D \tag{4.11}$$

That is, the duplicate statistic for two fields A and B is incremented by one, if one interval encloses another. The total number of such values, is collected as a statistic, by either empirically or by calculating an average using the probability of one interval randomly picked from a list enclosing another interval picked from same list (A. Here, we continue assuming that the statistics of U(O, A, B) and U(I, C, D) are precomputed and available. This cost component is roughly equal to total duplicate values in O that occur consecutively, multiplied by average number of duplicate values per record in I.

Average number of duplicates per record in I,

$$\frac{Size(I) - U(I, C, D)}{U(I, C, D)} \tag{4.12}$$

An approximation for consecutive duplicate records in O,

$$(runlength(O, A, B) - \frac{U(O, A, B)}{Runs(O, A, B)}) \times Runs(O, A, B) \tag{4.13}$$

So our final cost component is a product of 4.13 and 4.12.

$$\frac{Size(I) - U(I, C, D)}{U(I, C, D)} \times (runlength(O, A, B) - \frac{U(O, A, B)}{Runs(O, A, B)}) \times Runs(O, A, B) \tag{4.14}$$

A component yet to be figured out is k_{avg}. Finding k_{avg} is a challenge we will solve in next subsection. For now, the overall the cost component for I is the sum of eq 4.10, eq 4.8 eq 4.9 and eq 4.14.

$$\begin{aligned} C_{inner} = CF(I) + CF(I) \times Runs(O, A, B) + Runs(O, A, B) \times k_{avg} \times CN(I) \\ + \frac{Size(I) - U(I, C, D)}{U(I, C, D)} \\ \times (runlength(O, A, B) - \frac{U(O, A, B)}{Runs(O, A, B)}) \times Runs(O, A, B) \end{aligned} \tag{4.15}$$

Summing up the Costs

Hence, the formula for total cost is a summation of C_{inner} (4.15) and (C_{outer}) 4.7,

$$\begin{aligned} CF(O) + CN(O) \times Size(O) + CF(I) + Runs(O, A, B) \times k_{avg} \times CN(I) \\ + CF(I) \times Runs(O, A, B) \\ + \frac{Size(I) - U(I, C, D)}{U(I, C, D)} \times (runlength(O, A, B) - \frac{U(O, A, B)}{Runs(O, A, B)}) \times Runs(O, A, B) \end{aligned} \tag{4.16}$$

Finding k_{avg}

We know that the cost component in equation 4.10 should meet the criteria that the record r should be greater than or equal to the last record in the corresponding run in O. To put it mathematically,

$$\forall i, \ Max(Run_i(O_N)) \geq r \tag{4.17}$$

for i in range($i = 0$ to $Runs(I, C, D)$). Now, it may be the case that,

$$r = Max(I_N) \tag{4.18}$$

which means that r may well be the last record (k = Size(I)) in I. This means that the inner relation has read up to its last record for every run in O. Even though this is not a required condition, we will now see how an assumption makes our cost calculation better.

Uniformity of Runs Assumption

We have made an assumption regarding the distribution of records in outer file. It doesn't affect the end result or the cost formula. But construction of cost model and evaluation is significantly easier, because the required data set with the assumed distribution is easier to build and analyze. The assumption is that runs in O are distributed uniformly. It means a couple of things

- Each end of run occur in regular intervals after C records where C is a constant for the file. This constant is referred to as *runlength*. For example, a distribution of hundred numbers have uniformly distributed runs if a change in ordering happens only after 10 ordered numbers. Here, the average *runlength* is 10.
- An average run is a representative of the file itself in terms of the values. That is the first record in run is min(O) and last record is max(O). This need not hold true for every single run, but this is an attempt to estimate the distribution of an average run.

An approximation of runlength for in relation R is as follows,

$$runlength(R, A, B) = \frac{Size(R)}{Runs(R, A, B)} \tag{4.19}$$

But the assumption takes this a step forward and states that the runlength is also fixed for every run, rather than being an average value as a whole. As a consequence of our uniform runs assumption, we assume that equation 4.18 is true. In other words, according to our uniformity of runs assumption, the distribution of each run in O is a representative of the distribution of inner file I itself. Hence the inner will be scanned down until the end of the I for every run in O. Now, the cost component from equation 4.10 can be rewritten as

$$Runs(O, A, B) \times Size(I) \tag{4.20}$$

Note that, this assumption need not be true for the cost formula to be correct. But an extension of the cost formula takes advantage of this assumption and substitutes for k_{avg} although are ways by which we can evaluate k_{avg} without this assumption. Assuming uniformity runs assumption to be true is beneficial in another sense. The calculation of k_{avg} using the assumption would result in the cost formula calculating the worst case cost. But the true cost without the assumption is lower than the cost in 4.10.

Using equation 4.20, the overall cost in 4.16 is changed to,

$$\begin{aligned} C_{total} = {} & CF(O) + CN(O) \times Size(O) + CF(I) \\ & + Runs(O,A,B) \times Size(I) \times CN(I) + CF(I) \times Runs(O,A,B) \\ & + \frac{Size(I) - U(I,C,D)}{U(I,C,D)} \\ & \times (runlength(O,A,B) - \frac{U(O,A,B)}{Runs(O,A,B)}) \times Runs(O,A,B) \end{aligned} \tag{4.21}$$

Table 4.2 shows the cost components involved. To sum up, this cost formula is a sum of total number of times the records in O and I accessed when underlying fingered scan is in play factoring in repeated accesses due to multiple merges and duplicates.

Component	Cost
Finger Reset for O and I	$CF(O) + CF(I)$
getNext() costs for O	$CN(O) \times Size(O)$
getNext() issued for I per run in O	$CN(I) \times runlength(O,A,B)$
Total getNext() issed for I	$CN(I) \times runlength(O,A,B) \times Runs(O)$
Number of getFirst() called on I	$CF(I) \times Runs(O,A,B)$
Unique elements	$\frac{Size(I)-U(I,C,D)}{U(I,C,D)} \times (runlength(O,A,B)$ $- \frac{U(O,A,B)}{Runs(O,A,B)}) \times Runs(O,A,B)$
Adjustment factor	k_{avg}

Table 4.2: Cost Components in the Formula

4.2.1 About Special Cases

Now, we use the general formula we derived in 4.21 in two special cases. The two cases consists of a join operation on,

1. Two linked lists of records in memory
2. Two files in a disk and memory store where records are stored in files and files are accessed in pages loading them into memory

4.2.2 Cost of join on Two In-Memory Lists

Two in-memory linked lists present the simplest special case for the generic formula. The list of records now form a c-style linked list with a next field pointing to the address of next record. The generic formula can be converted this case if the two fields A and B in records are the same

$$A = B \tag{4.22}$$

for each record in both O and I. Hence the set (O, R, O.A, O.A, Runs(O)) and (I, R, I.A, I.A, Runs(I)) is involved in join. ie, the two fields we picked for the generic formula converges to one and that field forms part of join predicate. The cost formula remains exactly same, but the components involved in the formula, especially runs acquires the conventional meaning as Bradley explained in his work [2] in the light of equation 4.22.

Reduction to Sort Merge

A key criteria for the robustness of our cost formula when considered as an in-memory join is that, it should reduce down to the cost of a sort merge join in a situation where sort merge join would be picked by the optimizer. Optimizer picks sort merge join over others when both the tables involved in the join operation are sorted. This condition is simulated in our cost formula when the number of runs in the outer is 1. Inner is already totally ordered. Hence we have two totally ordered lists if,

$$Runs(O, A, A) = 1 \tag{4.23}$$

Substituting this in eq 4.21. The cost becomes,

$$= CF(O) + CN(O) \times Size(O) + CF(I) + 1 \times Size(I) \times CN(I) + CF(I) \times 1 \tag{4.24}$$

$$= CF(O) + CN(O) \times Size(O) + CF(I) + Size(I) \times CN(I) + CF(I) \tag{4.25}$$

CF(O) and CF(I) are insignificant terms in the equation as it is a constant. You can see that the equation reduces to

$$= CN(O) \times Size(O) + CN(I) \times Size(I) \tag{4.26}$$

This equation is similar to equation 2.5 which is the cost of merge. This equation also contains more CN terms. However, CN terms are simply a way to calculate the cost of a next operation which should essentially be included in equation 2.5 as well if calculated our way. Hence, the eq 4.21 has been reduced to cost sort merge join.

Reduction to Nested loop

Similarly, optimizer would pick a nested loop join over sort merge if the tables are completely unordered and our equation should be able to reduce to cost for nested loop in such a scenario. The worst case scenario in our case where O is totally unordered can be simulated if

$$Runs(O, A, B) = Size(O) \tag{4.27}$$

Let us substitute this in eq 4.21. The cost becomes,

$$CF(O)+CN(O)\times Size(O)+CF(I)+Size(O)\times Size(I)\times CN(I)+CF(I)\times Size(O) \tag{4.28}$$

removing the insignificant terms as in previous equation,

$$CN(O) \times Size(O) + CF(I) \times Size(O) + Size(O) \times Size(I) \times CN(I) \tag{4.29}$$

Note that the first two terms are also not significant as we have a term with $Size(O) \times Size(I)$ in it which outweighs lower powers, which results in the following equation,

$$Size(O) \times Size(I) \times CN(I) \tag{4.30}$$

This is the major component in the cost for nested loop join as mentioned in eq: 2.2.

4.2.3 Cost of Finger Enabled Join in a Disk and Memory Model

On calculating cost in terms of pages for a two level store involving a high speed memory and a low speed disk, we take certain liberties with the fields in records to conform it to our model. Consider a Join operation on two files (I & O) stored in a disk as pages and records per page. The operation works as follows:

1. A getFirst() is issued to the root of the operator tree.
2. A getFirst() is issued at the first page of the first column of the I. A finger (fi) is being initialized on the first page.

3. Page is loaded into memory.

4. A getFirst() is called on the first record of the page. A finger(fi1) is initialized on the first record in the page.

5. Steps 1-4 is repeated for outer file with another major finger (fo) on first record and a minor finger (fo1) on the first record.

6. As records are read from the first page of O, the minor finger successively moves to next records. Major finger points to the pageid. As long as records satisfy the condition 4.1, no attempt is made to reset the finger.

7. There are two possible outcomes in this scenario,

    - End of page occurs on O. The next page is loaded into memory and major finger is placed at the beginning of it. Minor finger is reset to the beginning of first record of second page.
    - A run occurs, ie, the condition referred in 4.1 is broken. In this case, the first page of I is loaded and hour hand of I is reset to the first page of I.
    - End of page occurs on I. Next page is loaded into memory and scanning proceeds.

Consider the page-wise join algorithm which is a slightly modified version of the algorithm mentioned in section 2.

foreach *page po in O* **do**
    foreach *page pi in I* **do**
        foreach *record r in po* **do**
            foreach *record s in pi* **do**
                if $r_i == s_i$ **then**
                    add $< r, s >$ to result;
                end
            end
        end
    end
end

Algorithm 3: Pseudo code for paged nested loop join

The two outer loops operate on a low speed disk and the two inner loops operate high speed in-memory. Compared to the cost of reading pages from disk, the cost of inner loop is free. Our model can capture this difference in speed and still use the finger optimization.

The same algorithm that was written above can be split into a two level join. Two inner loops that operate in memory can be considered as a fingered scan operation from memory. Two outer loops which essentially reads the pages from disk can be thought of as a "fingered page scan" operation from disk.

Cost Formula in terms of getNext() and getFirst() cost

The algorithm 4 is a page-by-page look at the data compared to a record-by-record look. The cost formula we derived for a record-by-record can be modified to suit the page by page version with a few tweaks. To facilitate this fingered page scan operation, we need additional statistics and a few assumptions about the optimizer. If we consider the two inner loops of the algorithm 4 as free, what remains is the double nested loop on pages. Such an algorithm looks like this,

foreach *page po in O* **do**
  foreach *page pi in I* **do**
    res = page_join(po, pi) ;
    add res to result;
  end
end

Algorithm 4: Pseudo code for paged nested loop join ignoring CPU cost

where page_join(po, pi) is considered as a free in-memory operation. You can see that this algorithm resembles our simple record-by-record join algorithm form section 1. This enables us to see this as a special case of our generic join.

The property of a run can be extended beyond individual in-memory list of records. But it is not just sufficient to think of pages as linked list. We need some kind of ordering within the pages. To facilitate this property in the linked list of pages, we store the min() and max() record of each page. A = Min() and B = Max() respectively as given in Figure 4.1. (Min(), Max()) naturally forms an interval. These two new statistics per page per field in the record helps to apply the generic formula to this special case. For a file R containing Size(R) records stored in i pages and each record containing N fields out of which two fields are (A, B), Let,

$$A = min_N(pr) \tag{4.31}$$

$$B = max_N(pr) \tag{4.32}$$

indicate the minimum and maximum value of Nth field of all the records contained in page R.These two new statistics is substituted for (A, B) fields. Hence, join operates on the sets,

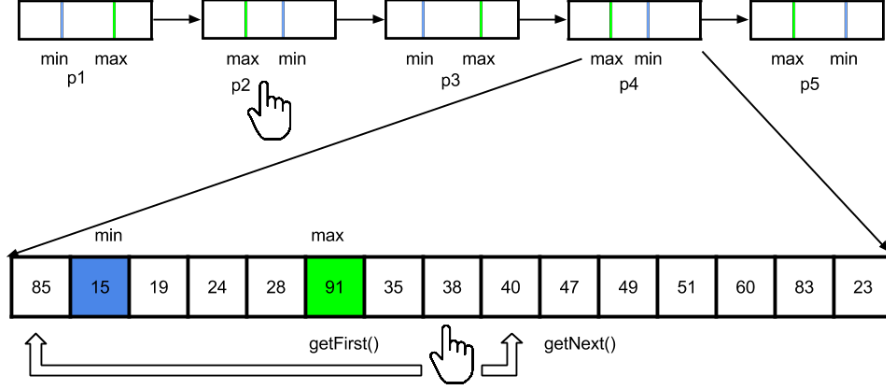

Figure 4.1: Fingered Join on a Disk and Memory Model

(O, R, O.Min(), O.Max(), Runs(O)) and (I, R, I.Min(), I.Max(), Runs(I)). The definition of run remains same as defined in equation 4.2. That is, we detect a run only if,

$$min_N(po_{i+1}) \leq max_N(po_i) \forall i \tag{4.33}$$

Hence, Runs(R) statistic is the total number of runs satisfying this criteria. Similarly others are slightly modified to reflect this. Table 4.3 shows the refined statistics.

Symbol	Name
$Size(R)$	Total number of pages occupied by file R
$CN(R)$	Cost of loading the next page in sequence
$CF(R)$	Cost of loading the first page in sequence
$runlength(R, min, max)$	Average runlength of N_{th} column in terms of number of pages
$min_N(Page_i(R))$	The minimum value of N_{th} field of all records in $Page_i(R)$
$max_N(Page_i(R))$	The minimum value of N_{th} field of all records in $Page_i(R)$

Table 4.3: Statistics Available for the Optimizer for Fingered Page Scan

At this point, we can use the exact same equation 4.21 for calculating cost.

$$CF(O) + CN(O) \times Size(O) + CF(I) + Runs(O, min_N(Page_i(R)), max_N(Page_i(R))) \\ \times Size(I) \times CN(I) + CF(I) \times Runs(O, min_N(Page_i(R)), max_N(Page_i(R))) \tag{4.34}$$

Table 4.4: Selectivity Estimates of Various Operators

Operator	Type	Selectivity Factor(F)
Cross Product	Binary	1
Union-All	Binary	N/A
σ Column = Value	Binary	$1/U(O,N)$
σ Column1 = Column2	Binary	$1/\text{Max}(U(O,N), U(I,N))$
Column >value	Unary	$\frac{Max_N(R)-value}{Max_N(R)-Min_N(R)}$
Column in range(value1, value2)	Unary	$\frac{value2-value1}{Max_N(R)-Min_N(R)}$
Projection	Unary	N/A
Duplicate elimination	Unary	$U(R,N)$ / Size(R)
Sort Ascending	Unary	1
Sort Descending	Unary	1
Difference	Binary	(U(O, N) - U(I, N)) / U(O, N)

4.3 Calculating Other Statistics in the Join Node

The previous section includes a calculation of overall cost for the operation. For a generic query optimizer with a cursor based implementation, a measure of overall cost for join is not sufficient. The situation considered above calculates cost for a simple join operation which is essentially a simple tree with two nodes and a root node. In order to generalize this, we need this tree to grow larger with more operations connected to it.

Every query plan presented in the form of such a tree with a cursor based implementation will have the equivalent of getFirst() and getNext() executed at various times in all nodes in the tree. This would also mean that, instead of having an overall cost in the root node, we need the same statistics defined in the join node in terms of statistics we have in the leaf nodes. The statistics we assumed that the optimizer will have are summarized in Table 4.1. In essence, we have to calculate the resultant size of the relation after join, resultant number of runs and a value for cost component CF() and CN() on the join node. Table 4.5 shows the resultant runs statistic and size of the output for various operators. Table 4.4 shows the corresponding selectivity estimates.

The statistics we still need to calculate include a value for CF() and CN() in the root node. We know that the overall cost calculated is a function of CF() and CN() costs on the root node. The total cost is comprised of exactly one getFirst() call and

Table 4.5: Runs and Size Estimates of Various Operators

Operator	Type	Resulting runs (Nth field)	Size of Output
$\sigma Column = Value$	Binary	1	$fracSize(R)U(O,N)$
Cross Product	Binary	$Size(I) \times Runs(O,N)$	Size(O) X Size(I)
Union-All	Binary	$Runs(O,N) + 1$	Size(O) + Size(I)
$\sigma Column1 = Column2$	Binary	$\frac{Runs(O,N) \times Size(I)}{Max(U(O,N),U(I,N))}$	$\frac{1}{/} Max(U(O,N), U(I,N) \times Size(O) \times Size(I)$
Column >value	Unary	$Runs(R,N)$	$\frac{Max_N(R) - value}{Max_N(R) - Min_N(R)} \times Size(R)$
Column in range(value1, value2)	Unary	$Runs(R,N)$	$\frac{value2 - value1}{Max_N(R) - Min_N(R)} \times Size(R)$
Projection	Unary	$Runs(R,N)$	Size(R)
Duplicate elimination	Unary	$U(R,N)$	$U(R,N)$
Sort Ascending	Unary	1	Size(R)
Sort Descending	Unary	Size(R)	Size(R)
Difference	Binary	$Size(O) - Size(I)$	$(U(O,N) - U(I,N)) \times \frac{Size(O)}{U(O,N)}$

$(Size(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) - 1)$ getNext() calls.

$$C_{total} = CF(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) + (Size(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) - 1) \times CN(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) \tag{4.35}$$

Considering an even distribution of cost,

$$CF(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) = \frac{C_{total}}{Size(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)})} \tag{4.36}$$

$$CN(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) = \frac{C_{total}}{Size(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)})} \times (Size(\overset{\rightarrow}{\bowtie}_{(A,B)=(C,D)}) - 1) \tag{4.37}$$

CF, CN costs along with the estimates mentioned in table 4.4 completes the cost calculation process giving us every tool required for estimating the cost. This information is sufficient for the optimizer to pick the right join method for the scenario.

Chapter 5

Evaluation

5.1 Introduction

The evaluation goal is to see if the cost predicted by new cost formula conforms to the empirically measured cost. We create two files O and I containing a list of records and perform a join operation between them. The join predicate is an equality on a key field and non key field. The presence of a key field in the operation ensures that we meet the criteria of having a completely ordered inner relation. Section 5.2 explains how the *data generator* create files to suit these requirements.

The output of data generator is passed through a join simulator which can perform both nested loop and finger enabled joins. Join simulator takes five inputs and returns two outputs. It takes paths to O and I and the field names on O and I to perform the join on along with the type of join that is required to be performed as input. Type of join can either be nested loop or finger enabled join. As output, the result set containing a list of items and a numerical cost value is returned.

We calculate two costs and plot them against each other. Empirical cost, returned by the join simulator and predicted cost is calculated using the cost formula. Conventionally, the empirical cost of join operation quantifies some form of disk I/O. One way of measuring it is by counting the number of disk pages read, assuming a constant number of records per page. There are a few other methods of assessing the cost, but the defining criteria is that cost should be directly proportional to the relative time various query plans takes to run. Such a measure would enable the optimizer to pick the fastest plan. We use sum of every accesses to every single record as such a measure of cost. A cost calculator module returns

the number of times every record is read by incrementing a cost counter on every access to a record. As the join operation progresses, the cost calculator module keeps global cost counter that gets incremented on each read operation. However, the cost calculator module is oblivious to the type of join being performed. This independence is important because a separation of cost calculation process and join process removes the possible tight coupling which could result in a false or a biased cost.

Second, we calculate the predicted cost from cost formula. The basic elements in the cost formula, the CF and CN variables, are substitutable to calculate the cost based on disk access speeds as well as a simulated cost for a single level store. Since we use a simpler evaluation model that uses the number of records as the unit of cost, the goal of substituting for CF and CN variables should be to arrive at number of records as cost.

Once both the costs are calculated, they are plotted against the *product of record size of O and I* for both predicted and empirical determined values. Both are plotted side by side in same graph to show the correlation between costs. The purpose of plotting against the product of cost is two fold. First, it removes the quadratic nature of nested loop costs from the graph. This removal makes easier to visualize the difference between predicted and empirical cost. Second, it ensures that both Size(I) and Size(O) are involved in visualization process.

5.2 Dataset Generation

Dataset generator takes a set of constraints as arguments and returns a file containing records that meets these constraints. The synthetic file generated this way contains a sequence of records each containing fixed number of user specified fields with positive integers as values. User can specify the following constraints as parameters passed down to the constructor of dataset generator.

- Number of fields per record
- Names of each field used as the identifier
- Number of runs in each field
- Size of file (number of records in the file)
- The fields to be finger enabled

- Primary key
- Total number of keys in every field

Every join process is preceded by a dataset generation phase which creates two files O and I meeting two user specified criteria mentioned above. The resultant dataset can be saved on disk for later joins.

5.3 Evaluation Overview

The files O and I are passed through a join simulator along with the type of join to be performed. For a standard experiment, we calculate four costs for two hundred join operations as data points. The four costs calculated per data point are,

cost1: Empirical cost of nested loop join by passing nested loop join as join type argument to simulator.

cost2: Predicted number of records scanned according to System R cost formula.

cost3: The same dataset is used to the empirical cost for finger enabled join to see the improvement.

cost4: Predicted number of records scanned after introducing runs statistic and fingered scan.

These four costs are plotted against each other in a predetermined order to show a sequence of steps,

1. cost1 is plotted against cost2 to show that a cost model inspired by System R adequately measures the cost of a nested loop join.
2. We then introduce fingered scan operator. Here, cost2 is plotted against cost3 to show that,
   - There is an improvement in performance as indicated by experimental cost of finger enabled join.
   - Due to the wide gap between the line segments, it is evident that the improvement in performance indicated by experimental cost is not adequately captured by System R style cost model.

3. Finally, cost3 is plotted against cost4 to show that the newly introduced statistic Runs(), when used as a function of cost equation can effectively capture the improvement in performance .

While a direct measurement of cost from our generic formula is non trivial due to the presence of CF and CN terms, there are ways in which total cost can be approximated by assigning a fixed cost to these operations. As mentioned earlier, we can decide what is returned by the cost formula by substituting appropriate values in place for CF and CN terms in the formula. Since we decided to go with number of records as the measure of cost, we substitute cost of getNext() (CN) as 1 unit and cost of getFirst() (CF) as 0 units to obtain a cost in terms of number of records read.

5.4 Experiments and Results

5.4.1 Experiment Overview

There are several variables in the cost formula. The variables that are relevant to us are the ones that show the comparison between predicted and actual cost at scale. We varied the record sizes of O and I sampling randomly from 200 - 10000 values for sizes. Number of runs in O was varied between 10% to 60% records. 200 samples are plotted in three graphs showing the cost against the product of Sizes of relations. Figure 5.4.1 shows the conventional System R cost formula plotted against observed cost of nested loop join. It can be seen that they conforms to each other as the line showing predicted cost closely follows the line showing empirical cost. Due to lack of index scan, scanning of individual files is equivalent to a segment scan in System R style. This results in a straight line segment in Figure 5.4.1 as predicted cost for System R.

In Figure 5.4.1, we plot the cost for fingered scanning of records. This results in a performance improvement and the empirical cost has gone down compared to previous graph. This reduction is significant as you can see from the graph. It is plotted against the predicted cost from System R to show that the System R cost model is incapable of predicting the cost advantage incurred using fingered scan. This is evident from the wide gap between both line segments in Figure 5.4.1. At this point, we introduce the runs statistic and build up a cost equation based on the formula studied in this thesis. The cost predicted by new formula using runs statistic is given in Figure 5.4.1. The experiment confirms that the new cost formula incorporates the improvement in performance as you can see that both the line segments closely follow each other.

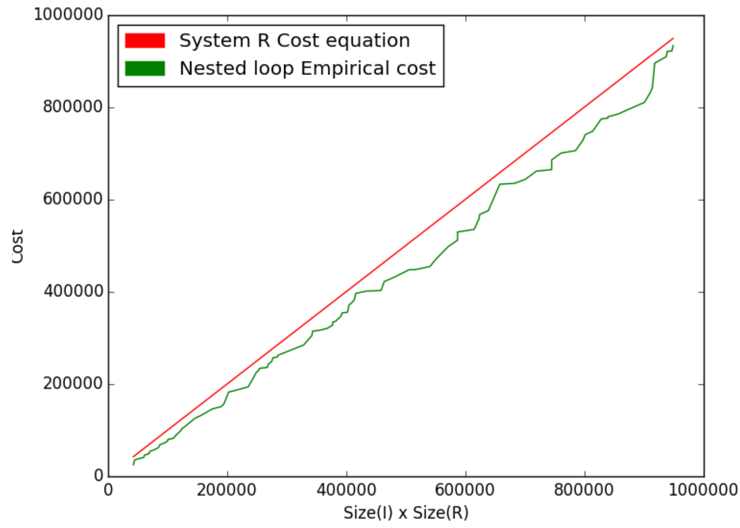

Figure 5.1: Experimental(Nested loop) vs Predicted(System R) values

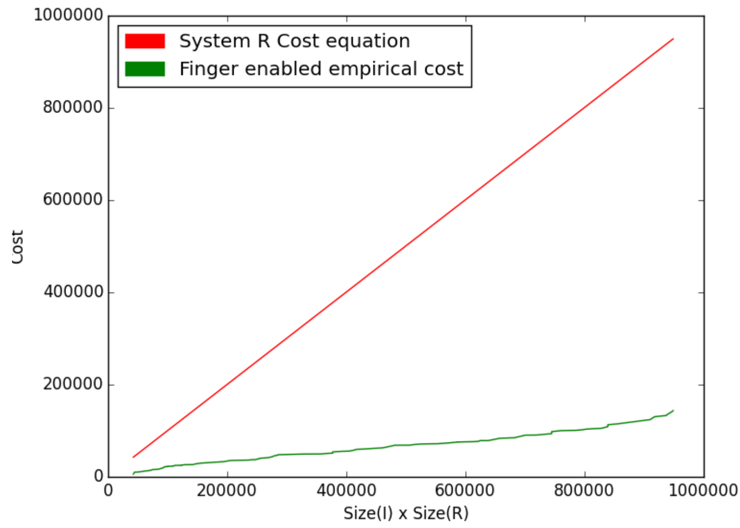

Figure 5.2: Experimental(Fingered Scan) vs Predicted(System R) values

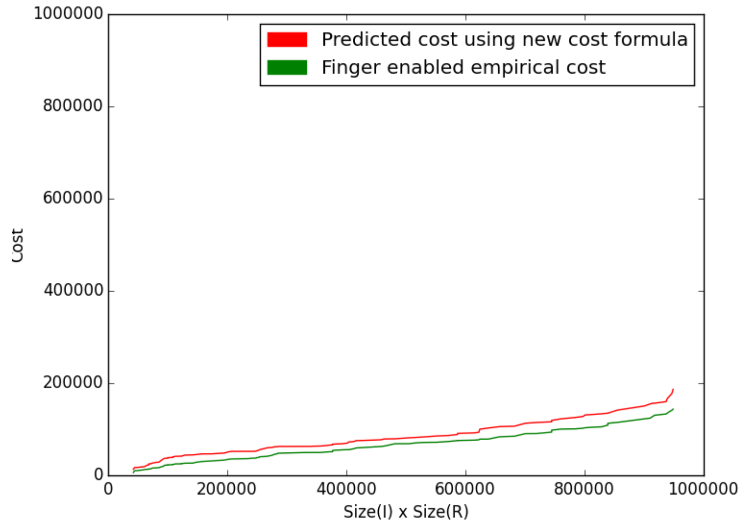

Figure 5.3: Experimental(Fingered Scan) vs Predicted(New cost model) values

5.4.2 Verifying and Explaining Anomalies

As observed from the graph, the result of experiment (actual value) closely corresponds to the predicted value. However, there are a couple of notable points from the Figure 5.4.1:

- The actual value is always slightly lesser than the predicted value.
- There is a divergence between the line that follows actual value and the one that follows predicted value as number of records gets higher an higher as the size of records grows.

Both of these two properties of graph can be attributed to the k_{avg} attribute we had considered during the construction of cost formula. k_{avg} stands for average records to be read in I per run in O. The disparity occurs as a consequence of the uniformity assumption we had discussed earlier. We had assumed that every run uniformly spans between minimum and maximum value of join field from I. That is,

$$\forall i \; Min(I) = Min(Run_i(O)) \tag{5.1}$$

$$\forall i \; Max(I) = Max(Run_i(O)) \tag{5.2}$$

However, this is not true in reality for every single run. While Equation 5.1 is not of consequence here, equation 5.2 needs to be fine tuned to work with edge cases. Maximum value of each run in O will fall short of maximum value of join field from I. That is, $Max(Run(O, A, B))$ will be less than $Max(I, C, D)$ for a large number of runs in O. This results in the finger in O never reading the last few records of I several times. This happens more and more as the number of runs increases and resulting in the widening lines. Fortunately, we don't need an exact answer to this question as picking a query plan is about approximation and our estimates without this approximation still estimates the cost reasonably well.

Chapter 6

Inferences and Future work

In this thesis, we study the cost model for fingered scan, a newly introduced method of scanning records. We show that fingered scan improves performance of join operation and this improvement in performance can be captured using a cost formula which incorporates runs statistic as a component.

6.0.3 What's not Covered?

We have not dealt with index scans which should ideally form an immediate future work. The finger based scans on B+tree style indexes would incur a different cost than segment scans of records. The cost considered in this model is more related to a segment style scan where records are scanned based on their consecutive positions in disk.

We also haven't dealt with finer details of query cost calculation like calculating the cost for nested queries, multiple joins, group by and order by clauses etc. Although such a calculation forms an important part of an overall cost plan, this work is only a first step in that direction. However, we do calculate various cost parameters for the join node which can be used for cost analysis in other places where finger enabled join is not used.

Although, the cost model studied in this thesis deals with the definition of runs that incorporates two fields A and B in a record, there are several other ways to add more information about the records. Even though those scenarios aren't dealt with in this thesis in detail, they are a good candidate for future work.

6.0.4 Knowing Other Information

In a DBMS system, there are different kinds of information that we can utilize in such a setting. This thesis lays out the basics of the idea, but applying this to some of the other statistics and constraints can produce better results. For example in the disk and ram cost model we discussed, if we know that,

$$r_{i+1}.A > r_i.A \tag{6.1}$$

while encountering a run, resetting the minor finger to the beginning of the page would be sufficient instead of loading from first page. Some such information we can use are,

- Functional dependencies
- Primary keys
- Number of unique elements
- Probability of an interval falling in another

More work is required to take better advantage of the fingered scan based on this new information and map it in to a cost formula.

Placing Multiple Fingers in I

Placing a third finger (other than regular and for handling duplicates) finger in I may enable us to go back only a certain number of records instead of rewinding to beginning of I on encountering a run in O. Knowing where to place the second finger requires collection of more distribution oriented statistics. In general, multiple fingers in I can improve the performance, but at higher cost of maintenance.

Appendices

Appendix A

Derivation of Probability of One Interval Falling in Another

Low and high values of A have the same probability, but high values of B are much more likely than low values) we have

$$p = \int_0^1 \int_0^1 \frac{1}{1-a} \int_a^1 \frac{1}{1-c} \int_c^1 f(a,b,c,d)\,\mathrm{d}d\,\mathrm{d}b\,\mathrm{d}c\,\mathrm{d}a$$

where

$$f(a,b,c,d) = \begin{cases} 1 & \text{if } a < c < d < b \text{ or } c < a < b < d \\ 0 & \text{otherwise} \end{cases}$$

We can rewrite this as

$$
\begin{aligned}
p &= 2\int_0^1 \int_a^1 \frac{1}{1-a} \int_c^1 \frac{1}{1-c} \int_c^b 1 \, \mathrm{d}d \, \mathrm{d}b \, \mathrm{d}c \, \mathrm{d}a \\
&= 2\int_0^1 \int_a^1 \frac{1}{1-a} \int_c^1 \frac{b-c}{1-c} \, \mathrm{d}b \, \mathrm{d}c \, \mathrm{d}a \\
&= 2\int_0^1 \int_a^1 \frac{\frac{1}{2} - c + \frac{1}{2}c^2}{(1-a)(1-c)} \, \mathrm{d}c \, \mathrm{d}a \\
&= \int_0^1 \int_a^1 \frac{1-c}{1-a} \, \mathrm{d}c \, \mathrm{d}a \\
&= \int_0^1 \frac{\frac{1}{2} - a + \frac{1}{2}a^2}{1-a} \, \mathrm{d}a \\
&= \frac{1}{2}\int_0^1 (1-a) \, \mathrm{d}a \\
&= \frac{1}{4}.
\end{aligned}
$$

References

[1] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[2] James V Bradley. Distribution-free statistical tests. 1968.

[3] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[4] Edgar F Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

[5] Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.

[6] Jim Melton. *Understanding the new SQL: a complete guide*. Morgan Kaufmann, 1993.

[7] Priti Mishra and Margaret H Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[8] Ari W Mozes. Method and system for histogram determination in a database, February 10 2004. US Patent 6,691,099.

[9] Viswanath Poosala and Venkatesh Ganti. Fast approximate query answering using precomputed statistics. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, page 252. IEEE, 1999.

[10] Viswanath Poosala, Venkatesh Ganti, and Yannis E. Ioannidis. Approximate query answering using histograms. *IEEE Data Eng. Bull.*, 22(4):5–14, 1999.

[11] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD Record*, volume 25, pages 294–305. ACM, 1996.

[12] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[13] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB JournalThe International Journal on Very Large Data Bases*, 6(3):191–208, 1997.

[14] Wikipedia. Cursor (databases) — wikipedia, the free encyclopedia, 2015. [Online; accessed 21-June-2015].

[15] J Wolfowitz. On the theory of runs with some applications to quality control. *The Annals of Mathematical Statistics*, 14(3):280–288, 1943.