# Compression in Memory Constrained DBMSs

**M. Tech Dissertation**

Submitted in partial fulfillment of the requirements
for the degree of

**Master of Technology**

by

**Ashwini G Rao**
**Roll No: 03305011**

under the guidance of

**Prof. Krithi Ramamritham**

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

# Acknowledgments

I am extremely thankful to **Prof. Krithi Ramamritham** for his guidance. His constant encouragement has helped me a lot. Working with him has been a joyful experience. I thank my parents and friends for their support and encouragement.

Ashwini G. Rao
IIT Bombay

# Abstract

Small devices like handhelds, personal digital assistants, cellphones are very popular today. They are used for simple personal applications like Ediary and complex enterprise applications like Health-care and Micro-banking. The complex applications are resource intensive and require storage management, query processing, transaction management and synchronization techniques. Query processing includes executing complex join and aggregate queries locally on the device. A lightweight DBMS is required on the device to support application requirements and to manage data effectively. The DBMS should use techniques that make best use of the limited memory and computation power of the small device.

Data compression techniques have been used to improve query performance in conventional database systems. They have been used in small device database systems to save space and communication costs. In this work we tried to find out whether data compression techniques can be used to improve query performance in small device database systems. We studied the problem of intelligently selecting data for compression, so that compression always speeds up queries. We show that a cost based approach that is aware of the query and device characteristics is required to properly reap the benefits of data compression. We propose a 2-phase cost based, compression performance and query work load aware optimizer for automatically deciding the compression techniques. A Simputer based implementation was used for validating the performance of our approach. The results of the performance analysis show that our approach leads to significant improvement in query performance and also provides good space savings.

# Contents

# Chapter 1

# Introduction

Handheld computing devices are very popular today. The convenience of low cost, light weight, small size and adequate computing power has led to their popularity. An example is the Simputer [Sim], developed by IISc, Bangalore and Encore Software. It has 32-bit Intel Strong Arm processor running at 206MHz, 16-64 MB SDRAM, 08-32 MB Flash Memory for permanent storage, a $320 \times 240$ monochrome or LCD display panel, a stylus for input and Linux operating system. It has support for wireless communication, Internet access and smart cards. Its cost ranges from Rs 10000 to Rs 20000. HP iPaq and Palm PalmOne are other examples.

## 1.1 Need for DBMS on Handheld

Due to the *Anywhere*, *Anytime* convenience, handheld devices are being used for both personal and enterprise applications like Ediary, Health-care and Micro-banking. These are complex applications and require effective management of data. Some of their requirements are:

1. Management of fairly high volume of data.
2. Simple select, project and join queries and also complex aggregate queries.
3. Sophisticated access rights management using views and aggregate functions are needed to protect data privacy.
4. ACID properties of local and global transactions have to be maintained.
5. Periodic synchronization with remote server is required when data is downloaded from the remote server and processed offline.

A database management system on the handheld can effectively meet all these requirements.

## 1.2 Handheld DBMS versus Disk DBMS

Database implementation techniques for disk based, powerful computers are well known. The same techniques can not be used in handhelds due to reasons like:

- Handheld devices have small memory, computing power and communication bandwidth compared to these high-end systems.

- The database in the handheld device will be stored in flash memory. The characteristics of flash memory are very different from that of magnetic disks. Flash memory read time is small compared to disk read time. Techniques used in disk based systems assume data to be primarily disk resident. The main optimization criterion for query optimization algorithms, buffer pool management, indexed retrieval techniques, transaction management and other techniques is minimizing number of disk accesses.

- Handheld operating systems (e.g., PalmOS, Windows CE ) do not offer the same variety of services as desktop operating systems. For example, PalmOS does not support threads or processes for background tasks, a common technique for desktop computer applications.

- Transaction management and synchronization techniques in a handheld DBMS have to consider issues like mobility and frequent disconnection of handhelds.

- Better security measures like encryption of stored data are essential in handheld devices as they are easily stolen or lost. Techniques like compression of data for saving storage space and communication costs are more relevant in handhelds. These aspects have to be considered while choosing the above techniques.

## 1.3 Compression in Databases

Compression is a commonly used technique in many areas. Some examples are compression of video and audio files for data transfer, compression of files for backup, tarred or gzipped software distribution packages. Compression techniques exploit the redundancy present in representation of data. The redundancy is in the extra bits used to represent data or the repeated occurrences of a given sequence. By removing redundancy, the same data can be stored in lesser space. Compression is generally used to reduce storage and communication costs. From a general DBMS perspective compression has the following advantages:

- Storing data and indexes in compressed form saves disk space and may improve buffer utilization.

- The disk input and output time will reduce as smaller blocks of data have to be read from and written to the disk.

- Log files will be shorter when the log records are stored in compressed form.

The disadvantages are as follows:

- Compression is CPU intensive. During query processing, the compressed data has to be decompressed. Decompressing large amount of data may lead to increase in query processing time. Insert, delete and update operations will also involve compression and decompression of data.

- The effectiveness of error detection and correction techniques depend on the amount of redundancy present in the data representation. Reducing the redundancy in data representation will decrease the effectiveness of these techniques.

In disk based DBMS, disk space is cheap. The use of compression to save space at the cost of increase in query processing time is not recommended. However, recent studies [Ray95, HRS95, GRS98, HWKM00] have shown that use of lightweight compression in a read intensive, disk based DBMS can lead to more efficient query processing. This is possible as the major cost involved in query processing, i.e., the disk read time, reduces due to the use of compression. The reduction in disk read time will compensate for the compression and decompression overhead.

## 1.4 Problem Definition

Applications for PDAs execute complex join and aggregate queries on the handheld device [BBPV00]. Most of the modern day cellphones are witnessing complex data centric applications being developed for them. Sensor networks are also coming up rapidly and these collect data from the environment and subject them to various queries. Most of these queries need to be executed on the device itself to reduce communication costs.. Thus, there is an increasing need to facilitate execution of complex queries locally on a variety of lightweight computing devices.

The query optimizer should generate efficient query execution plans for the complex queries. It should exploit all aspects that can help in generating optimal plans. Efforts have been made to make the optimizer aware of the available memory and the underlying storage model [Sen04]. Another aspect that may be useful is compression as it has been used to speed up query execution in disk based DBMS. In disk DBMS a set of heuristics has been used to decide what data to compress. The following questions sum up our problem definition. *"Can compression be used to speed up query execution is a handheld DBMS?"*, *"Are heuristics sufficient to fully reap the benefits of compression?"* and *"If heuristics are not sufficient, how do we decide what data to compress?"*.

## 1.5 Road Map

In this chapter we looked into the need for a DBMS in a handheld, use of compression in databases and our problem definition. The rest of the report is organized as follows: Chapter 2 discusses the general compression techniques and design issues. Chapter 3 looks into the issues involved when using compression in handhelds. Chapter 4 proposes the model for query performance aware database compression and presents the results of performance evaluation of the proposed model. Chapters 5 and 6 briefly discuss the issues in transaction management and synchronization in handhelds. Chapter 7 discusses the Database system developed. Chapter 8 presents the summary and future work.

# Chapter 2

# Compression

In this chapter we briefly discuss some of the common compression techniques that can be used for database compression. It is followed by a brief discussion of the design issues involved in selecting compression techniques. Detailed information about compression techniques can be found in [Sol00].

## 2.1 Compression Techniques

Compression techniques can be broadly categorized as follows.

### 2.1.1 Lossy and Lossless

Lossy data compression leads to better space gains with reduced accuracy of data. Decompression in this case will not give back the original data. Lossy compression is generally applied to digitized voice and graphic images. Lossless compression retains the original data after any number of compression and decompression cycles. Compression of text files, executables etc. that can not tolerate loss of information use lossless techniques. Since the database we are considering is a textual database, only lossless compression is considered.

### 2.1.2 Dictionary Model and Statistical Model

The dictionary based schemes maintain a dictionary of commonly used sequences and output shorter codes for them whenever they are encountered. Statistical schemes compress the data by assigning shorter codes to more frequent characters.

#### General techniques

The following are some of the general compression techniques in use. Run length encoding (RLE), Arithmetic coding and Huffman coding are statistical compression techniques. Lempel-Ziv (LZ) is a family of dictionary compression techniques.

**DBMS specific techniques**

Some DBMS specific compression techniques are COLA [HRS95], Hierarchical dictionary encoding (HDE) [CGK01], FOR compression [GRS98] and NULL suppression [HWKM00]. COLA is a statistical technique and HDE is a dictionary based technique. Both are used for text compression. FOR and NULL suppression are used for numeric compression.

### 2.1.3 Adaptive Model and Nonadaptive Model

Nonadaptive technique is a two pass algorithm. In the first pass, data is scanned to collect statistics required for compression. The statistics are stored in decode or frequency tables. In the second pass, data is compressed depending on the gathered statistics. The same statistics are applied during decompression and future compression and decompression cycles. Adaptive technique is a one pass algorithm. Initially it assumes equal probabilities for all the characters or strings. The statistics are gathered as the data is compressed. The required decode table is built as the data is decompressed.

## 2.2 General Design Issues

Some of the issues that need to be considered while selecting a compression technique are discussed below.

### 2.2.1 Adaptive versus Nonadaptive

As discussed earlier compression techniques can be adaptive or nonadaptive.

- In a nonadaptive technique the space gain depends on the correctness of the statistics. Updates may change the actual data statistics drastically leading to a sharp drop in the space gain. In such a situation, data has to be rescanned and fresh statistics gathered. The already compressed data has to be reorganized according to the new statistics. Reorganization requires costly flash memory writes. It is important to note that flash memory write time is comparable to disk memory write time and also that flash memory can be written to only a limited number of times. It is better not to use nonadaptive technique when the frequency of updates is high. Scanning a large amount of data to gather statistics may be expensive. To reduce this cost, sampling can be done. Sampling will reduce the space gain, but will be faster.

- In an adaptive technique the decode table is built as the data is decompressed. Hence, decompression has to always start from the beginning of the compressed data block. Random access to data is not possible. This implies that the query execution time may increase a lot if many large blocks of data have to be processed. Usually, adaptive technique has higher space gain and higher query processing overhead compared to nonadaptive technique.

### 2.2.2 Granularity of Compression

In a relational database each relation is usually stored in a separate file. Each file is divided into fixed size pages. A page has a page header and stores several records. Each record has several attribute values and it can be fixed size or variable size depending on the support available in the DBMS. This model leads to four granularities of compression: file, page, record and attribute. Granularity affects the following:

- Space gain
- Query processing overhead
- Size of meta data required to track variable sized items

Generally, higher space gain is achieved when a large block of data is compressed. This is especially true when adaptive technique is used. An adaptive technique gathers statistics as it compresses data. In a larger block, there is higher probability of the same string repeating many times.

Query processing overhead is closely linked to granularity of compression. All levels of granularity except attribute level will involve decompression. The larger the amount of data to be decompressed, the higher the query processing time. The amount of data to be decompressed depends on whether the decompression can begin at any position in the compressed block, i.e., whether random decompression is possible. If random access is not possible then the decompression has to begin from the start of the compressed block. Query processing may become very costly when it involves many large sized compressed blocks and random decompression is not possible. Figure 2.1 shows the space gain and query performance of the different levels of granularity in a memory constrained environment.

Attribute level compression can allow query processing without decompression. Compression technique that allows query processing on compressed data is called **precise** technique. Consider a relation *Doctor* with an attribute *Doctorid*. Let attribute level compression be employed on the relation. Consider a select query that wants to retrieve all tuples with *Doctorid* equal to 10. The select query can first compress the value 10 and compare the compressed value to the values in *Doctorid* column instead of decompressing every value in the *Doctorid* column. A join involving two attributes compressed using the same statistics will not involve any decompression. Attribute level compression allows Lazy decompression[HWKM00]. Algorithms to do direct string pattern matching on compressed data [FAB94] are available. These can also be used to reduce query processing overhead.

In many compression techniques fixed size data items become variable sized after compression. There are several methods to keep track of the start and end positions of the variable sized data items and each method will consume extra space. This extra space is called **pointer overhead**. Coarse granularity results in lesser number of variable sized items and hence smaller pointer overhead.

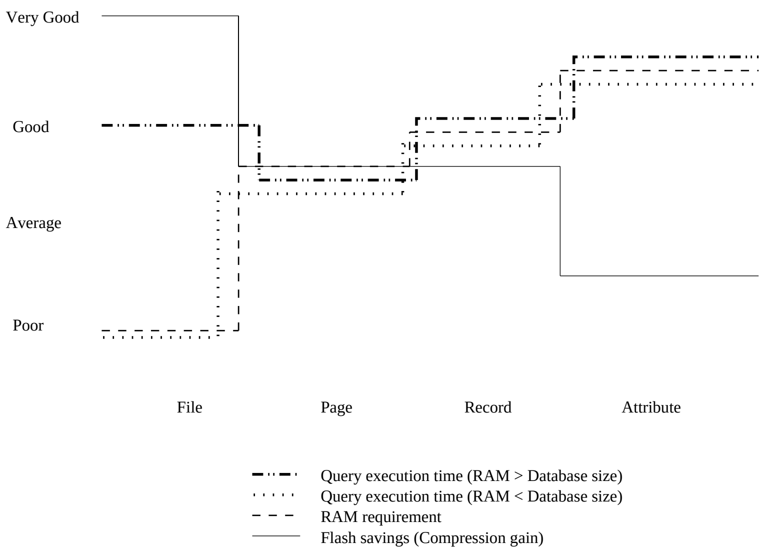



Figure 2.1: Performance at Different Compression Granularity

### 2.2.3 Lightweight Compression Techniques

Common compression methods like LZ, Huffman coding and arithmetic encoding have high decompression time. The performance difference between LZ and simple methods, like offset encoding (encoding a numerical value as as the offset from a base value) is an order of magnitude [CGK01]. The decompression time is an important factor from the view point of query processing in databases.

### 2.2.4 Preserving Lexical Order

The DBMS has to support operations like select, project, join and update. It also has to support aggregate and sorting operations if required. These operations require comparison of one or more attributes of a given relation with attributes of another relation. Comparison can be of two types, exact or range. In exact comparison, a compressed attribute value of one relation can be directly compared to another compressed value of another relation if both of them are compressed using the same statistics. Range comparison is more complicated. The compression technique has to preserve the lexical ordering of the data. Compression techniques that preserve lexical ordering of data when compressed are known as order preserving techniques.

# Chapter 3

# Compression in Handheld DBMS

In this chapter we will try to answer the first two questions, *"Can compression be used to speed up query execution in a handheld DBMS?"* and *"Are heuristics sufficient to fully reap the benefits of compression?"*, that we raised in Section 1.4. Section 3.2 discusses the motivation for using compression. In Section 3.2 we briefly discuss the assumptions we have made about the query processing environment in a memory constrained DBMS.

## 3.1 Memory Constrained Query Processing Environment

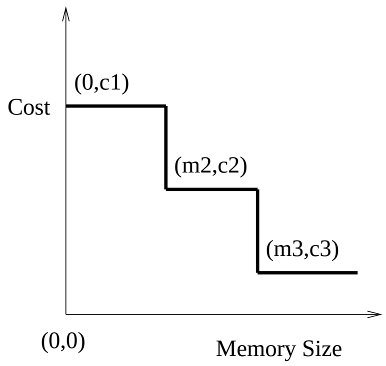
Lightweight computing devices are characterized by a small amount of main memory and most of them use flash memory as secondary storage. Writes to flash memory are very costly. Hence, the query processing schemes minimize materialization in secondary storage. Flash memory read is fast. So, flash memory is usually used as read buffer. Unless read/write ratio is very high main memory is used as write buffer. Avoiding materialization in flash memory limits the query execution capabilities in a small device.

### 3.1.1 Features of Query Optimizer

Some assumptions we have made about the query optimizer are discussed below.

#### Cost based Query Optimization

A cost based approach to query optimization is used as it leads to the best query execution plan. Suppose there are $n$ schemes $s_1, s_2......s_n$ to implement an operator $o$. Each scheme uses different amount of memory and has a different cost. Cost of an operator scheme is the expected time of computation that depends on the number of tuples, read times, index and aggregate list creation times, and lookup times. Each operator has a (memory, cost) profile associated with it depending on the memory requirements and execution times of its schemes. An example of a profile is shown in Figure 3.1. These profiles are constructed using cost formulae for the schemes. Queries are optimized based on such profiles. The operator schemes are discussed in detail in Appendix A.




(a)

Figure 3.1: (Memory, Cost) Profile of an Operator

**Left Deep Query Plan**

Since query processing should minimize writes to flash, materialization of intermediate results are minimized. Materialization if absolutely necessary, is done in main memory. A left-deep tree [SKS02] is used as the query plan tree since all other query plans resort to some materialization in flash. The left-deep tree is most suited for pipelined evaluation, since the right operand is always a stored relation, and thus only one input to each operator needs to be pipelined.

**Memory Cognizant Query Optimization**

When a fully pipelinable schedule is used, the entire memory is not available for each of the operators [HSS00]. Memory has to be shared optimally among all the operators. How much memory each requires depends on how each database operator is evaluated. The amount of memory available in handhelds is increasing with every new device and different handhelds come with different memory sizes. To use the resources as efficiently as possible, query plans are chosen depending on the amount of main memory available. The memory cognizant optimization can be 1phase or modified 2phase [Sen04]. A 1phase optimizer is more complex than a modified 2phase optimizer.

## 3.2 Motivation for Compression

Research [HRS95, GRS98, HWKM00, CGK01] has shown that compression can be used to improve query performance in Read-intensive Disk-based database systems. It motivates us to find out whether compression can be used to achieve similar results in a Memory-constrained database system. The results of our initial experiments showed that compression can indeed be used to improve query performance. Some salient results are shown in Figure 3.2 and

Figure 3.3. Our experimental setup is briefly explained below.

| | NLJ | INLJ |
|---|---|---|
| VISIT, PRES (PRES.VISITID compressed) | 31 - 32 | 1.2 - 1.3 |
| VISIT, PRES (Uncompressed) | 20 - 21 | 1.3 - 1.4 |

Figure 3.2: Performance (in *sec*) of Join Operator schemes on VISIT and DOCTOR

| | NLJ | INLJ |
|---|---|---|
| VISIT, DOCTOR (DOCTOR.DOCNAME compressed) | 3.0 - 3.3 | 0.1 - 0.2 |
| VISIT, DOCTOR (Uncompressed) | 3.2 - 3.5 | 0.1 - 0.2 |

Figure 3.3: Performance (in *sec*) of Join Operator Schemes on VISIT and PRES

### Experimental Setup

We run our DBMS on the Simputer[Sim]. It is a low cost handheld device produced in India. Our version of the Simputer runs on a Intel Strong Arm processor at 200MHz. It has 28MB of Flash memory for permanent storage and 24 MB of DRAM for main memory. The gcc cross compilation toolkit for Strong Arm processor was used to cross compile the code for the Simputer. Simputer runs the Linux operating system and other computation intensive applications.

We use the Health-care database schema and dataset of [BBPV00]. The schema is given in Figure 3.4. The dataset is a relatively large one (Doctor(91), Drug(77), Prescription(2155), Visit(830)). We consider complex join and aggregate queries. The joins are only on integer data. Selections are on both integer and string data. Flat storage model is used. Lightweight compression techniques like FOR[GRS98] and HDE[CGK01] are used.

| |
|---|
| Doctor(*docid int, name char[20]*) |
| Prescription(*visitid int, drugid int*) |
| Visit(*visitid int, docid int, date int*) |
| Drug(*drugid int, type char[20]*) |

Figure 3.4: Health-care Database Schema

Our experiments indicate that compression is useful if used properly. In the example in Figure 3.2 the query execution time has increased for the operator execution scheme NLJ after the integer join attribute PRES.VISITID is compressed. Whereas the query execution time of the operator execution scheme INLJ on compressed data is similar to the query execution time on uncompressed data. In the example in Figure 3.3 the query execution time for the operator execution scheme NLJ has decreased after the string attribute DOCTOR.NAME is compressed, but the query execution time for the operator execution scheme INLJ has not changed. These examples show that general heuristics like using compression for integers are not always beneficial. A more cost based approach is required to decide what relations or attributes to compress so that the execution time of the queries likely to be executed on them reduces. We consider the problem in the context of a fixed set of queries and propose a greedy model for determining what data to compress for a given set of queries in the next chapter. We discuss some of the insights we got from our experiments in the following subsections.

### 3.2.1 Cost of Compression

We define the cost of compression as the difference between the extra time required for decompression and the reduction in IO time due to compression of data. The major factor that decides the cost of compression is the operator schemes in a query execution plan. The decompression time depends on the total number of decompressions and the algorithm used for compression. The total number of decompressions required is determined by the execution schemes used for the operators. A better execution scheme has lesser number of decompressions. For example, hash join has lesser number of decompressions than nested loop join. A better compression algorithm has faster decompression. The IO time depends only on the number of reads from the flash as there are no flash writes involved. The total number of reads is determined by the execution schemes used for the operators. By compressing the data, the read time decreases. If the right balance is maintained between the reduction in execution time due to reduced flash read time and increase in execution time due to extra decompression time then overall query execution time reduces.

### 3.2.2 Query Execution Plan Structure

The query execution plan in a handheld DBMS is fully pipelined. The operators in the query plan share the results through very small buffers. The shared buffer usually stores a single result tuple[BBPV00]. Hence, compression does not lead to significant change in RAM usage. This implies that compression does not change join order of the query execution plan. The memory requirements of the join operator execution schemes are also not expected to change much as in handheld databases the join queries are mainly on integers and the in-memory indexes are built on integers.

### 3.2.3 Decompression Strategy

A decompression strategy determines when and where the compressed data is decompressed in the query execution plan. The strategy may be *eager decompression*, *lazy decompression* or *transient decompression* [CGK01, HWKM00, GRS98]. Eager strategy decompresses the data when the data is brought into main memory. In lazy decompression, data stays compressed during query execution as long as possible and is decompressed explicitly when an operator requires it. In transient decompression, an operator temporarily decompresses data, but keeps it compressed in the output.

In a disk based environment, compression of data may lead to a better operator execution scheme [CGK01]. To exploit this fact, [CGK01] uses the transient decompression strategy. It should be noted here that in our case transient operator concept is not applicable as compression cannot lead to a better operator execution scheme. In the query execution environment we consider, there is only one result tuple shared between the operators in the left deep query tree. Further reduction in main memory usage due to compression of data is very less. Also, in [CGK01] pipelined evaluation is not used. The result of every operator is written to the disk if its size exceeds the main memory size. Allowing the data to stay compressed reduces the time required for reading it back from the disk. We also do not consider eager strategy as it rules out direct processing on compressed data and also may lead to unnecessary decompressions. We consider the lazy decompression strategy to be most suitable for a memory constrained environment. We assume that the cost of a operator scheme does not increase if any attribute that is not involved in its execution is compressed.

### 3.2.4 Granularity of Compression

Our experiments showed that attribute level and record level compression schemes give good performance in a memory constrained environment. Furthermore, a combination of both better results in certain cases. The performance when using coarser granularity schemes like file level depend on the main memory available for caching the decompressed results. Each decompression of a large chunk of data takes substantial time. Caching results of decompression reduces repeated decompressions of the same data. Performance of attribute level, record level and file level compression for query *Q1* of the Health-care database is shown in Figure 3.6, Figure 3.7 and Figure 3.8. The performance of query *Q1* on uncompressed data is shown in the Figure 3.5. The execution times in the graph were got by a combination of implementation and emulation. Parts of the implementation were hard coded to handle query *Q1* only. Details about query *Q1* are in Appendix B.

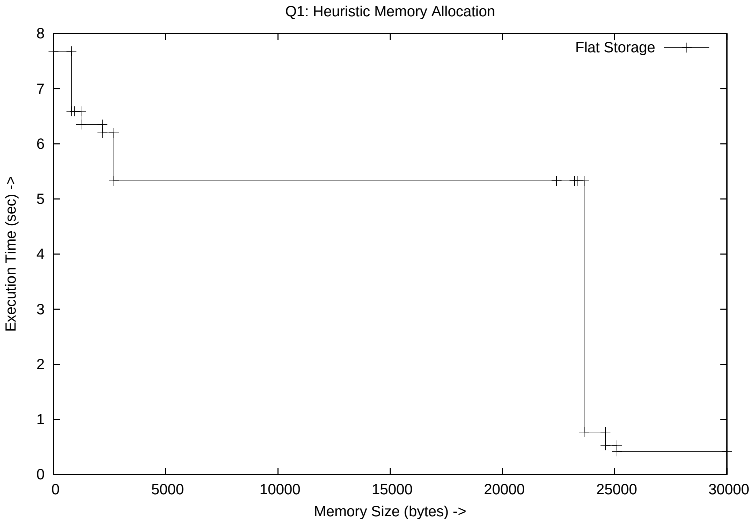



Figure 3.5: Performance of Q1 without compression

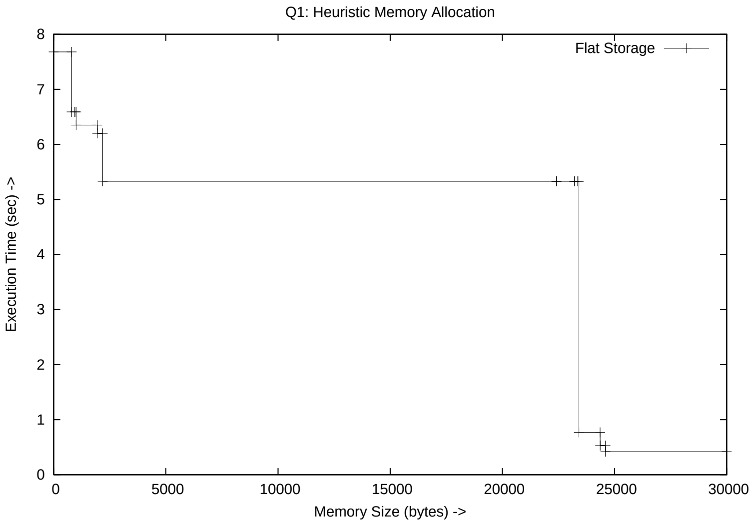



Figure 3.6: Performance of Q1 with attribute level compression

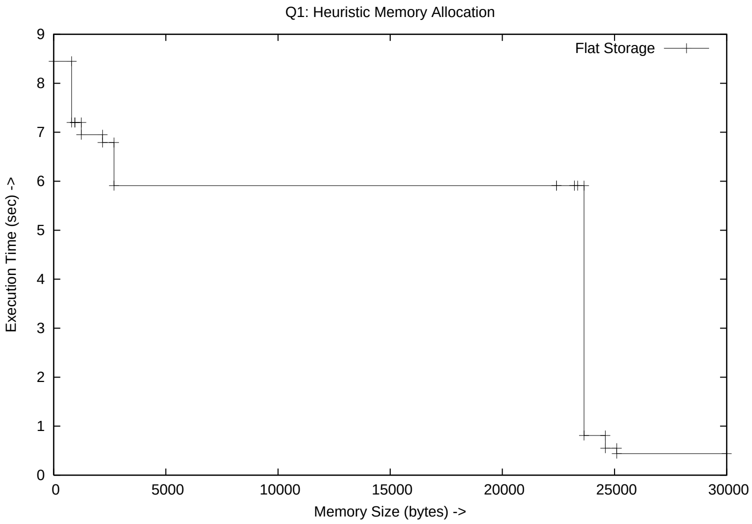



Figure 3.7: Performance of Q1 with record level compression

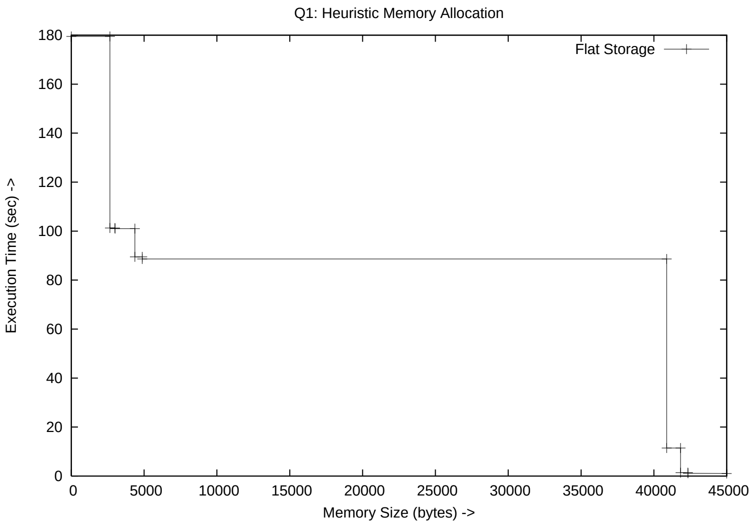



Figure 3.8: Performance of Q1 with file level compression

# Chapter 4

# Query Performance Aware Compression

In this chapter we will present a model that can be used to determine what data to compress for a given set of queries. Section 4.1 defines the benefit measure and the compression schemes for the operator evaluation schemes. Section 4.2 explains the model. Section 4.3 presents the results of the performance evaluation of the proposed model.

## 4.1 Compression Benefit

We define the benefit of compression as the change in execution time due to compression of the attribute or record. Positive benefit indicates a probable decrease in execution time and negative benefit indicates a probable increase in execution time. As explained in Section 3.2.1, in a memory constrained environment the cost of compression is determined by the operator execution schemes.

$Attribute\,Benefit = Execution\,Time\,before\,Attribute\,Compression -$
$Execution\,Time\,after\,Attribute\,Compression$
$Record\,Benefit = Execution\,Time\,before\,Record\,Compression -$
$Execution\,Time\,after\,Record\,Compression$

Each operator scheme executes on one or two attributes depending on whether it is a single input or two input operator. We introduce the concept of attribute benefit and record benefit for an operator scheme. Attribute benefit is the benefit got by compressing any of the input attributes of the operator scheme and record benefit is the benefit got by compressing the input relation at record level. Record benefit is computed only if the input is a base relation. We define the following parameters.

$dtime$: Decompression time. It depends on the compression algorithm used.
$m$: Number of right input tuples to an operator

$n$: Number of left input tuples to an operator
$m_1$: Number of times the right input to the operator is read from flash.
$m_2$: Number of times the right input to the operator is decompressed.
$n_1$: Number of times the left input to the operator is read from flash.
$n_2$: Number of times the left input to the operator is decompressed.
$avg\_rs$: Average size of the record in the right relation before compression.
$navg\_rs$: New average size of the record in the right relation before compression.
$avg\_ls$: Average size of the record in the left relation before compression.
$navg\_ls$: New average size of the record in the left relation after compression.
$fread$: Flash read time per byte of data. Depends on the kind of flash used.
$fseek$: Flash seek time. Depends on the kind of flash used.
$d$: Number of distinct groupby values.
$sub$: Cost of the subtree below the aggregate operator in the query plan

| Scheme | Left Input | Right Input |
|---|---|---|
| NLJ | $n$ | $n * m$ |
| BSTINLJ | $n$ | $m + n$ |
| HJ | $n$ | $n * (\frac{m}{b}) + m$ |
| Select | *none* | $m$ |
| Project | *none* | $m$ |
| Display | *none* | $m$ |
| NLASum | *none* | $(1 + d) * m$ <br> $+d * sub^*$ |
| BASum | *none* | $m$ |

Figure 4.1: Number of Reads for Operator Schemes

$*$ subtree below the aggregate operator is executed $d$ number of times. Each operator scheme in the subtree is executed d number of times. The read count of each operator scheme is incremented $d$ number of times.

To decide whether to compress at attribute level or record level, we need to compute the combined attribute and record benefit. Benefit can be estimated by finding out the total number of times the attribute or record is read from flash and decompressed. The total number of reads and decompressions depends on the operator schemes. We compute the attribute and record benefit of each operator scheme and sum them up to get the combined attribute and record benefit. To compute the compression benefit of an operator execution scheme, we estimate $n_1$, $n_2$, $m_1$ and $m_2$. $n_1$ and $m_1$ are estimated using the number of times each operator scheme is executed and the number of reads of the operator scheme. The total number of times each operator scheme is executed in a query tree is computed by traversing the tree. The number of reads for operator schemes is given in Figure 4.1. The number of inputs to the operator scheme, the requirements of the compression technique and the operator scheme determine $n_2$ and $m_2$. It should be noted that the number of reads and decompressions for an operator scheme is considered to be zero if the attribute or record was already accessed and decompressed in the query execution plan. After estimating $n_1$, $m_1$, $n_2$ and $m_2$ the compression benefit is computed using the formulae in Figure 4.2.

| | Compress None | Compress Left Input | Compress Right Input |
|---|---|---|---|
| Execution Time | $(n_1 * avg\_ls * fread)$ $+(m_1 * avg\_rs * fread)$ | $(n_1 * navg\_ls * fread)$ $+(n_2 * dtime)$ $+(m_1 * avg\_rs * fread)$ | $(n_1 * avg\_ls * fread)$ $+(m_1 * navg\_rs * fread)$ $+(m_2 * dtime)$ |

Figure 4.2: Execution Time before and after compression

## 4.2 Proposed Model

We propose a 2-phase optimizer to automatically select data for compression given a query workload. In the first phase we optimize each query in the workload using a memory aware query optimizer. In the second phase we compute the attribute and record benefit using the query execution plans from the first phase and decide to compress depending on the computed benefits. The rationale behind using a 2-phase approach is the assumption that data compression in a handheld database will not change the query execution plan structure. This point was explained in Section 3.2.1.

The main algorithm *GetIdealPhysicalLayout* is given in Figure 4.3. The input to the optimizer is a set of relations R in the database, their statistics, a set of queries Q and the required device specific parameters P. The device specific parameters include $fread$, $fseek$ and RAM size. The optimizer computes each attribute benefit and record level benefit. For each query in the query set Q, the optimizer gets a optimal execution plan using a memory cognizant optimizer. It is to be noted that multi-query optimization is not applicable as there is no materialization of intermediate results in flash. The optimal plan for a query indicates the execution schemes for the operators in the query. The optimizer computes good estimates of $m_1$, $m_2$, $n_1$ and $n_2$ for each operator execution scheme by traversing the query tree. For each operator scheme in a query, the optimizer computes the left attribute, right attribute and record benefit. It adds the benefit to the respective attribute benefit and record benefit. The procedure is repeated for all the the queries in the given set. Data is chosen for compression on decreasing order of positive benefit. A relation is compressed at record level if the record benefit is more than the combined benefit of all the attributes in the relation.

### 4.2.1 Time and Space Complexity

The functions $GetSchemeCount$ and $GetEstimate$ require traversing each query tree constant number of times. Time complexity of tree traversal is O($l$), where $l$ is the number of nodes in the tree and space complexity is a constant. We have used a $O(n2^{n-1})$ time and $O(2^n)$ space memory aware optimizer proposed in [Sen04]. $n$ is the number of operators. We have considered fixed number of compression techniques. Hence, the time complexity of our greedy algorithm is $O(kn2^{n-1})$, where $k$ is the number of queries in the given query set. The space complexity is $O(2^n)$

```
Inputs: Set of relations, R, Statistics of relations, S, Set of queries, Q, Device specific parameters, P
Output: Ideal physical layout
Initialize each record and attribute benefit
for each query q in Q do
  Get the optimal query execution plan qep for q
  qep = GetOptimalQEP(q, S)
  for each scheme s in qep do
    Compute the total number of times scheme is executed
    scount(s) = GetSchemeCount(qep, s)
    Estimate m1, m2, n1, n2
    (n1, n2, m1, m2) = GetEstimate(qep, s, S)
    Compute the left attribute and right attribute benefit
    (labenefit, lrbenefit) = GetAttributeBenefit(qep, s, S)
    Compute the record benefit for each input base relation
    rbenefit = GetRecordBenefit(qep, s, S)
    Add the benefit to the corresponding attribute and relation
    TransferBenefit(s, labenefit, rabenefit, rbenefit)
  end for
end for
Compute combined attribute benefit for a relation
Compress relation at record level if record benefit is more than combined attribute benefit
```

Figure 4.3: Algorithm: ***GetIdealPhysicalLayout***

```
Inputs: Query, Q, Statistics of relations, S
Output: Optimal query execution plan
Use a memory cognizant optimizer to get the optimal query execution plan
Return optimal query execution plan
```

Figure 4.4: Algorithm: ***GetOptimalQEP***

```
Inputs: Query execution plan, qep, Execution scheme, s, Statistics of relations, S
Output: Attribute benefit
Compute left attribute benefit
labenefit = GetBenefit(s, qep, S)
Compute right attribute benefit
rabenefit = GetBenefit(s, qep, S)
Return labenefit, rabenefit
```

Figure 4.5: Algorithm: ***GetAttributeBenefit***

**Inputs:** Query execution plan, $qep$, Execution scheme, $s$, Statistics of relations, $S$
**Output:** Record benefit
Compute the record benefit
$rbenefit = GetBenefit(s, qep, S)$
Return $rbenefit$

Figure 4.6: Algorithm: ***GetRecordBenefit***

## 4.3 Performance Analysis

In this section we present the results of performance evaluation of our model. We carried out the evaluation on three query sets that are in increasing order of complexity. The details of the query sets are given in Figure 4.3, Figure 4.3 and Figure 4.9.

| | |
|---|---|
| Set1 | Complex Joins |
| Set2 | Complex Joins + Selects |
| Set3 | Complex Aggregates |

Figure 4.7: Complexity of Query Sets

| Query Set | Query (Scheme) | Frequency |
|---|---|---|
| Set1 | Q5 (NLJ NLJ INLJ)<br>Q5 (INLJ INLJ INLJ)<br>Q5 (NLJ INLJ INLJ) | 10<br>5<br>15 |
| Set2 | Q1 (NLJ NLJ NLJ)<br>Q5 (NLJ NLJ INLJ) | 10<br>10 |
| Set3 | Q4 (BA NLJ NLJ NLJ)<br>Q4 (BA NLJ NLJ INLJ)<br>Q4 (NLA NLJ HJ INLJ) | 10<br>5<br>10 |

Figure 4.8: Query Sets

### Choice of Compression Techniques

We have used the FOR[GRS98] compression technique at page level for attribute level integer compression and LZW for record level compression.

We have used a modified version of HDE[CGK01] for string compression. The entires in the dictionary are stored in a separate sequential file in flash. The position number of the dictionary entry is used as the key. The key value of first entry is 1, second is 2 etc. Number of bytes required to represent each key value is equal to $(\frac{number\ of\ keyentries}{256})$. The dictionary entry can be the whole attribute or a smaller chunk. The number of dictionary entries is equal to the number of distinct chunk values. One key entry per chunk has to be stored for every attribute value in the relation. We consider building the dictionary for chunk size equal to attribute size or half the attribute size. This can, however be extended to chunk size that is any multiple of a particular value. We compute the total decompression time required for each chunk size considered and use the chunk size that has minimum decompression time. The decompression time is given by the formula $(n_{read} * n_{key} * key_{size} * fread) + (n_{decomp} * n_{key} * fseek) + (n_{decomp} * size_{attr} * fread)$ where $n_{read}$ is the number of times the attribute is read, $n_{key}$ is the number of key values per attribute value, $key_{size}$ is the number of bytes

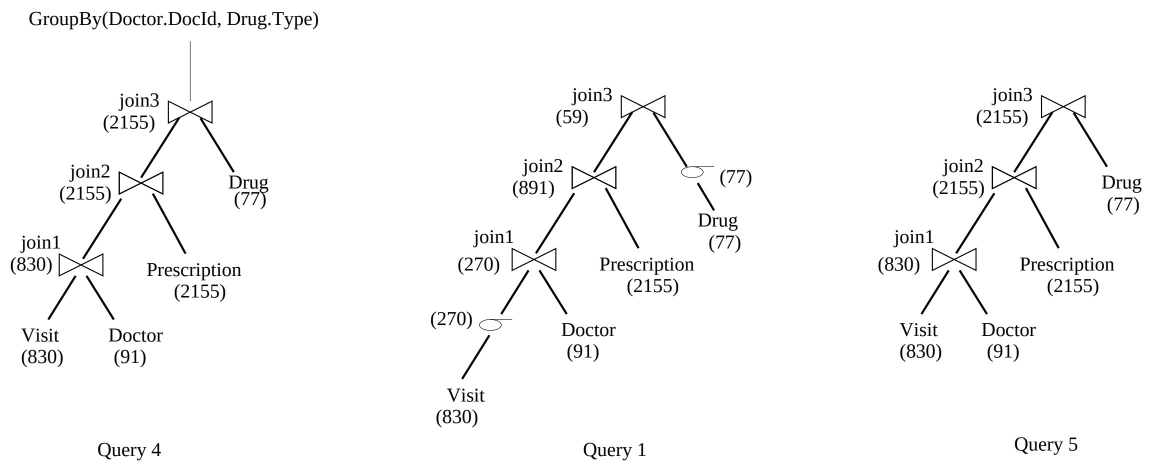



Figure 4.9: Query Structure of Q4, Q1 and Q5

required for each key value, $n_{decomp}$ is the number of times the attribute is decompressed, and $size_{attr}$ is the original number of bytes in each attribute value. At present, sharing of dictionaries among different attributes and caching of dictionary in RAM are not considered.

### Results and Observations

The compression gain, that is the decrease in overall query execution time, obtained for the three query sets after compressing the data indicated by optimizer is shown in Figure 4.10 and Figure 4.3. It can be noted that the gain decreases as the complexity of the queries, that is computational complexity increases. The gain predicted by the optimizer and the actual gain are indicated in Figure 4.3. The gain predicted is quite precise for Set1 and Set2. Set3 involves complex aggregate queries and it is difficult to estimate the parameters for aggregate queries. Hence, the gain predicted is not very precise. For Set1 the optimizer indicated the string attributes DOCTOR.NAME and DRUG.TYPE to be compressed. For Set2 and Set3 optimizer indicated the attribute DOCTOR.NAME to be compressed. The attributes DOCTOR.NAME and DRUG.TYPE are decompressed only when the results are displayed for the queries in Set1. As can be noted from Figure 4.9, the number of result tuples are small compared to the number of tuples in the relations. Hence they are being decompressed small number of times. In Set2 and Set3 the attribute DRUG.TYPE is involved in select and groupby operators and is decompressed large number of times. Hence, optimizer does not indicate DRUG.TYPE to be compressed. Compressing attributes NAME AND TYPE using Dictionary encoding reduces the record size of the corresponding relations drug and doctor from 24 bytes to 5 bytes. The decompression time for Dictionary encoding is computed by considering the number of *fseek* and the amount of data read from the dictionary.

We did not find any benefit when integer was compressed at attribute level using the page level FOR technique even though it lead to large decrease in average record size and integer has small decompression time. Investigation revealed that the code handling page level FOR compression has to handle the different sizes of compressed values in different pages and this adds extra time overhead which nullifies the benefit. The optimizer has to account for the time overhead of the code to accurately estimate the benefit of compression. The code for FOR technique at file level is simpler as it has to handle only a fixed length. We expect it to give better benefit.

The maximum flash space savings that can be obtained by the compression techniques that we have used is shown in Figure 4.3. We compressed only the attributes DOCTOR.NAME and DRUG.TYPE. The flash savings obtained by compressing them is indicated as the actual savings availed.

To check whether adding compression techniques to the executor engine added extra overhead, we ran a representative set of queries on uncompressed using both the unmodified executor engine and the compression aware executor engine. The performance of both were similar.

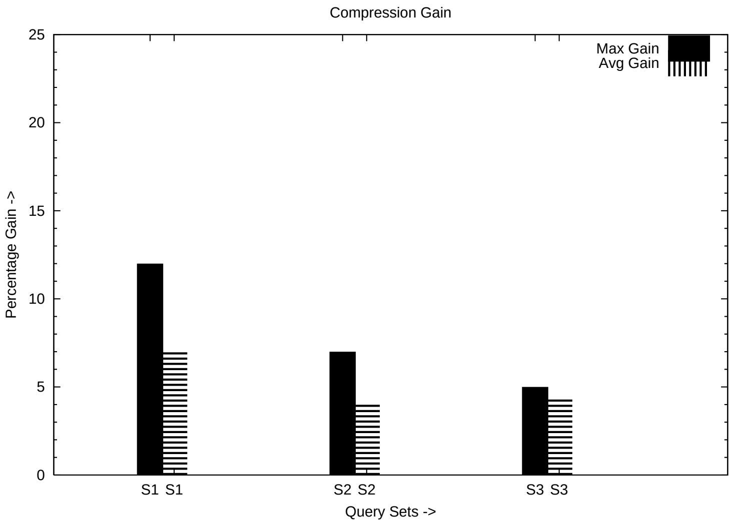



Figure 4.10: Compression Gain for S1, S2 and S3

| Query Set | Max Gain | Avg Gain |
|---|---|---|
| S1 | 12% | 7% |
| S2 | 7% | 4% |
| S3 | 5% | 4.3% |

Figure 4.11: Compression Gain for S1, S2 and S3

| Query Set | Compressed Attributes | Max Predicted Gain | Max Actual Gain |
|---|---|---|---|
| Set1 | DOCTOR.NAME<br>DRUG.TYPE | 51.5 | 44.2 |
| Set2 | DOCTOR.NAME | 27.1 | 23.0 |
| Set3 | DOCTOR.NAME | 1570.0 | 228.4 |

Figure 4.12: Gain Predicted by Optimizer

| Relation (size in bytes) | Max Saving Possible | Actual Saving Availed |
|---|---|---|
| Drug (1848) | 56% | 56% |
| Doctor (2184) | 19.5% | 0 |
| Pres (17240) | 74.5% | 0 |
| Visit (9960) | 49.4% | 0 |

Figure 4.13: Flash Space Savings

# Chapter 5

# Transaction Management

The DBMS on the handheld has to execute local transactions and participate in global transactions. It has to ensure the ACID (atomicity, consistency, isolation and durability) properties of transactions.

## 5.1 Issues

1. The frequent disconnections of the handhelds have to be captured by the transaction techniques.

2. Transaction management techniques should make optimal use of the memory, processing power and communication capabilities of the handhelds.

3. Mobility of the handheld device has to be considered.

4. Handhelds are easily lost, stolen or damaged. Hence, to ensure durability the changes have to be written to a server in the fixed network. Usually the data in the handheld is a copy of the data in some server. We can say that the outcome of the transaction has become durable only after the changes have been reflected at the server.

## 5.2 Concurrency Control

Concurrency support in handhelds depends on two factors.

- Handheld operating system: Handheld operating systems like PalmOs do not support multi-threading or multiprocessing. In such a case concurrency control is not required.

- User requirements: When multi threading or multiprocessing is supported by the operating system (e.g., Linux OS in Simputer) tasks can be allowed to execute concurrently in case the designer of the handheld wishes. In the Simputer the user can access the address book and simultaneously run mp3 music player in the background. Consider the example of running AQUA[Aqu] on the Simputer. AQUA is an on-line discussion forum for animal and crop diseases. It is being developed in Media Labs Asia, IIT Bombay.

To serve the farmers in the villages it has been ported to the Simputer. The required data from the main AQUA data server can be downloaded into the database in the Simputer. Once the data is downloaded into the Simputer the farmers can access the data without being connected to the main server. Periodically the data in the Simputer database must be synchronized with the data in the main AQUA data server. Ideally this should be done automatically without the farmers' interference. This will ensure that the synchronization is done even in the absence of the user and also the farmer need not bother about the technical details. The synchronization task can execute as the farmer is trying to access the data. A long duration task like data aggregation may also run simultaneously. It may or may not access the data accessed by the other two tasks. Local concurrency control is needed if we want to prevent dirty reads and writes when data is shared among concurrent transactions.

A very simple concurrency control mechanism should be sufficient in a handheld DBMS. Very few concurrent processes are present. Most of the time they will be accessing different data. In rare cases two tasks may need the same data. The situation described above is one of them. We can use 2PL[SKS02] with coarse granularity locks. Locking can be done at the table level. This will allow processes that are accessing different data to continue execution and will have lesser execution overhead. In the above example using table level locks will allow the data aggregation task to execute if it is accessing a different table. We will not deal with global concurrency control or isolation across transaction executing in different handhelds.

## 5.3 Atomicity

Atomicity is the all or nothing property. Local atomicity and Global atomicity in a handheld database are discussed in detail below.

### 5.3.1 Local Atomicity

Consider updating an entry in the address book of the Simputer. The user may update the email address and the residence address. Local atomicity ensures that either both the fields are updated or none are updated. Either in place update or shadow based update can be used to ensure local atomicity.

- Shadow based update is rarely used in disk based databases as it changes disk locality and also difficult to implement when concurrent transactions are present. Creating a new copy of the updated object in another place in the disk changes the disk locality. Change of disk locality will increase the number of disk accesses. The disk locality problem does not exist in a flash memory based database. Using shadow paging will simplify recovery as redo operations are not required. The main drawback of shadow based update technique is that it is poorly adopted to pointer based storage models like Ring storage. The object location changes with every update. If the size of the shadow object is large then the location of many other tuples will also change. If the size of the shadow object is small then the system page tables will have more entries. In either case the number of writes will increase.

- In place update uses write−ahead logging (WAL). The old values are stored in log records. In place update is better suited as it accommodates pointer based storage and its cost is insensitive to growth of flash memory storage. However, recovery is more complicated. The buffer replacement strategy will be *Steal*[SKS02] as the main memory in the handheld is very limited. Undo will be required if the dirty blocks have been written to handheld stable storage. To avoid Undo we can store the dirty blocks in some other stable storage. The smart card when attached to the handheld can provide this stable storage. We can output the dirty blocks to the stable storage of the smart card. No Undo will be required if the transaction whose dirty blocks have been written to the smart card is aborted. In rare cases, the smart card may be pulled out by the user before the transaction completes. Recovery will have to be initiated. One more optimization is to use pointer logging. Pointer to the values can be logged instead of the actual values[BBPV00]. Logging pointers will shorten the log record length and save space.

### 5.3.2 Global Atomicity

Consider an epurse application. A given amount of money has to be transferred from a bank account to an electronic purse in a smart card. The smart card will be attached to a device like Simputer that can communicate with the bank's server. The transaction involves three devices. Atomicity at the global level is ensuring that the changes are reflected at all the three devices or at none. The most common way to ensure global atomicity is to use the Two phase atomic commit protocol (2PC)[AGP98]. 2PC consists of two phases. The first phase is the *Voting* phase, and the second is the *Decision* phase. In the Voting phase the coordinator of the transaction verifies that participants involved in the transaction can ensure the ACID properties of their part of the transaction. In the Decision phase, the coordinator will send the commit or abort decision to the participants. 2PC has several shortcomings when used in the handhelds. The two rounds of messages exchanged can have a high communication cost. The large number of forced writes has a high write overhead. 2PC requires that the handhelds remain connected during the voting phase and the decision phase. Many optimizations to 2PC like Presumed Commit[AGP98] and Presumed Abort[AGP98] have been proposed. Presumed Commit reduces the cost associated with committing the transaction and Presumed Abort reduces the cost associated with aborting the transaction. The advantages of One phase commit (1PC) have also been explored in the mobile, disconnected environment[BPA00]. One phase commit eliminates the voting phase. If the participants can always ensure the ACID properties of their part of the transaction then there is no need for the voting phase. The participants are always ready to commit. The most constraining requirement of 1PC is that the participants' transactions are cascade less and realizable. Using strict 2PL[SKS02] will ensure this. It is not always possible to use strict 2PL. Weaker levels of isolation [SKS02] are sometimes used in databases to enhance efficiency. It has been proved in [AGP98] that if the databases satisfy certain conditions then 1PC can work correctly with weak levels of isolation. In the application AQUA explained in Section 5.2, all the databases are under our control. Hence, 1PC can be used. If external databases that are not under our control are involved then 1PC cannot be used. When the handheld supports smart cards, 1PC is useful. Smart cards satisfy all the constraints required by 1PC[BBPV00]. Hence, transactions involving only the handheld and the smart card can use 1PC.

## 5.4 Consistency

Local consistency can be maintained by providing techniques to define integrity constraints. Global consistency control is discussed in the next chapter on synchronization.

## 5.5 Durability

Durability property ensures that the result of a transaction is made permanent. Either the changes are written to permanent storage like flash memory before the transaction completes or enough information about the transaction is logged in permanent storage to enable the database system to reconstruct the updates after a failure. Handhelds can be easily lost or damaged. The data on the handheld is usually a copy of the data in a remote server. In such a case enough information about the changes done to the local copy has to be sent to the remote server or at least to another server that can communicate with the remote server. This will ensure that the changes are not lost in case of damage to the handheld or loss of the handheld.

In most 1PC protocols the co-coordinator will log all the participants' updates before triggering the atomic commit protocol. Thus, using 1PC ensures durability along with atomicity. Using logical logging or operation logging can save log space,but recovery can become more complicated[SKS02]. As explained in Section 5.3.1 Pointer based logging can be used. As explained in the chapter on synchronization, a handheld database transaction may use tentative commit. The transaction can actually commit only after the transaction is executed at the remote server, which may take a long time. In such a case alternative log maintenance techniques can be explored. For example, in Extended ephemeral logging[KD97] separate log queues are maintained for short duration and long duration transactions. The time and space requirements of such a scheme has to be analyzed in the case of a handheld.

# Chapter 6

# Synchronization

Users want to access data anytime and anywhere. Storing the data from the remote server in the handheld is convenient for the user. User should be able to query and also update this data. Some example scenarios where offline data processing and subsequent synchronization are useful are described below:

1. Scenario 1: A Mobile Salesperson

   The archetype of the mobile device user is a salesperson on the go. A favorite saying in sales is that "nothing happens until something gets sold." If that is true, then we should do whatever it takes to get sales data back as soon as it can be made available. In addition, we want to send data out to a mobile salesperson−order status, leads, and other critical sales data−to help focus their efforts on the things that really matter to a business. Since most salespeople rely on their cell phones, when they are equipped with a handheld device it seems natural to connect it to the phone for data transport. Between sales visits, a salesperson can be on the cell phone talking to a client. At the same time, the device can talk to the home office via its cellular connection−uploading order information and downloading new data points from headquarters.

2. Scenario 2: The Wireless Warehouse

   Some devices will never wander far from home, but will still need to be mobile within a limited geographical area. An example is a bar code reader. Barcode readers are commonly deployed in warehouses for inventory, order tracking, and other database-centric needs. Often, the barcode readers are on a wireless RF network. This allows data to be moved back and forth between the central data store and individual devices as it's read. If the barcode readers serve as dumb terminals and the central servers perform all processing then the central server can become swamped during peak work hours when all the devices clamor for its attention. Also, from time to time the network might be unavailable, either because a device is out of range or because the network radio signal is blocked by some physical obstruction. Deploying a database on the device is an ideal solution because each barcode reader can operate independently while its user roams the warehouse. All the data that's needed can be kept on the device itself, in the database. Periodically, when the device is in range and the server is available, the database on the device can be synchronized with the central data store. Such a configuration provides a maximum of flexibility, with continual updating of all relevant databases.

## 6.1 Issues

1. To maximize the availability of data in presence of disconnections replication can be employed. Handhelds can be allowed to independently update the replicated copy. This can lead to conflicts like:

   - Update-update conflict: Simultaneous updates to the same row and or column at two places
   - Update-delete conflict: Update to a row at one place and deletion of the same row at another
   - Unique or primary key violation: Simultaneous inserts of rows that have same value on a unique key value.
   - Failed change conflict. An integrity constraint may be defined on the data at one device and it may not be defined at another. The device where the integrity constraint does not exist can make changes which cannot be enforced at the device where it is.

2. Some kind of global consistency has to be maintained between the various replicated copies. The various possibilities are described below.

   - Data is partitioned between the copies: Strict consistency can be maintained between the copies. The application in the handheld can update the data and commit. The changes have to be propagated to the server.
   - Data is rarely shared between the copies: Strict consistency can be maintained by using reservation protocols[BPA00]. Part of the data can be reserved for a particular copy for a given period of time, that is, it can be leased. A lease is like a lock applied on the data. The application can obtain reservation on the data it requires from the server. It can use the data and commit the changes. The changes have to be propagated to the server.
   - Data is commonly shared between the copies: If data is shared between many copies and they try to access it concurrently then reservation protocols can become too restrictive. Maintaining strict consistency in the absence of reservation protocols will require that all copies be connected, which is not possible in case of handhelds. We can only have a relaxed correctness model like Eventual consistency[TTP+95]. Applications can read and write the data independently, but they can not commit the changes as there may be conflicts. Only tentative commits are possible. The

actual outcome of the transaction is known only when the transaction is executed at the main server.

3. When independent updates are allowed, the copies have to be periodically synchronized by merging the changes with the main server. Conflicts have to be detected during synchronization and detected conflicts have to be resolved. Conflict detection and resolution should be application specific for maximum flexibility. The system should provide methods for the application to specify conflict detection and conflict resolution procedures.

4. Ideally the synchronization protocol must be device, back end and network agnostic. It should be extensible and inter operable between heterogeneous data sources. For example, strings can be represented using different character encodings in different machines. In such a case these strings can not be directly compared. It is difficult to keep track of encoding used by different devices. Ideally the synchronization must take care of converting the local representation to a common representation like unicode. Similarly a XML like protocol can be adopted for data types like integer.

5. Conflict detection requires tracking of changes made at each copy. To track changes, there should be way to uniquely identify the rows. For example, unique row identifiers can be assigned to the rows at the server. A log of the updates, deletes and inserts has to be maintained. Tracking and propagating changes incrementally is required as handheld devices have small communication bandwidth and communication cost is high.

6. The user should be able to download a relation, part of a relation or the result of a query on to the handheld. Storing only a part of the relation or the result of a query can save storage space in the handheld. It can also save communication costs. Storing only a part is like defining a view. All the problems associated with view update translation [Lan90] are applicable. This will not pose any new problems if eventual consistency is used. The changes are tentative and maintained in a separate log. The changes to the view can be rejected later if it introduces inconsistencies at the device from where it was downloaded.

## 6.2 Existing models

Existing implementations [Ham, SG00, OLi, Act] for handhelds are based are based on the Publish Subscribe model and the Personal computer (PC) to Handheld synchronization model. The Publish Subscribe model is a three tier approach with one or more backend servers, middle layer support servers and several handhelds. A PC to Handheld synchronization model is a two tier model with a single personal computer and a single handheld device.

# Chapter 7

# The Database System

The directory structure of the code base developed is illustrated in Figure 7.1. Some important points about the implementation.

- The Executor engine follows the *lazy iterator* model. It has support for query execution on compressed data. *Lazy decompression* strategy is implemented.
- The Storage manager provides support for compression and decompression of data at record level and attribute level. The required compression and decompression techniques have been implemented.
- The proposed optimizer model is implemented.
- Support for transactions was implemented as part of the MTech project.
- A Synchronization tool was developed as part of the MTech project.

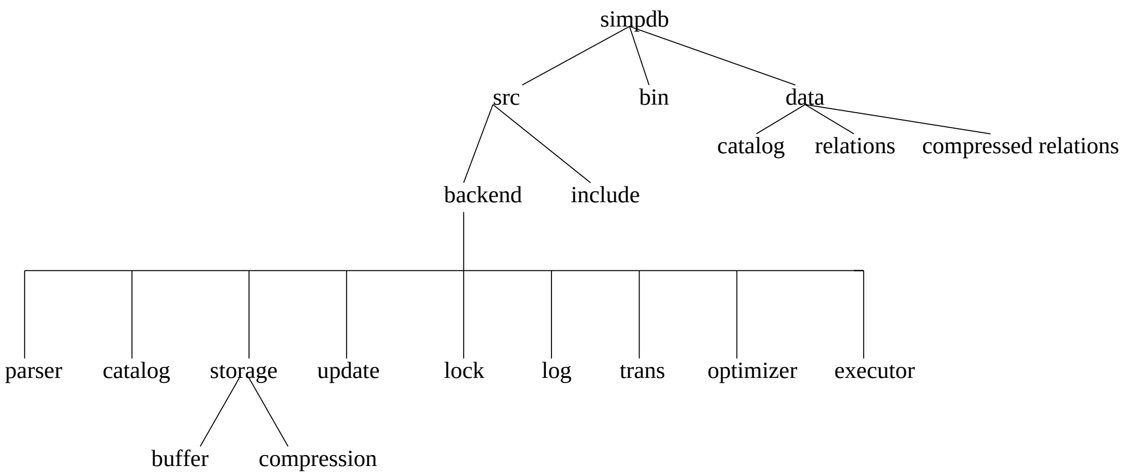



Figure 7.1: Code Directory Structure

# Chapter 8

# Summary and Future work

We studied the need for a DBMS on a memory constrained device and noted that the application requirements include executing complex queries on the device itself. The need to execute complex queries motivated us to explore techniques that can effectively speedup or improve query performance. We explored the usefulness of executing queries on compressed data and found out that it does give significant benefits. We also found that heuristics are not sufficient to properly exploit the benefits of using compression. We proposed a greedy, cost based optimizer to decide what data to compress given a set of queries that are likely to be executed on the device. The performance evaluation of our model showed our approach leads to significant benefits. Our contributions are summarized below.

### Contributions

- Showed that lightweight data compression techniques can be used to achieve significant speedup for a given workload in constrained devices.
- Showed that a cost based approach is required even when choosing integer data for compression.
- Proposed a 2 phase compression aware optimizer that will automatically choose data for compression given the query workload and device specific parameters.
- Showed the effectiveness of our proposed approach through a performance evaluation.

As part of the MTech project work was also carried out on issues in transaction management and synchronization in handheld DBMS. A transaction manager and Synchronization tool were implemented as part of this work.

### Future Work

In this work we have not considered materialization of intermediate results in flash as flash writes are costly. Intermediate results can be, however, materialized offline, i.e, when the ideal physical layout is decided for the given set of queries. This is same as creating views.

The query aware optimizer will then have to look into compressed views and view sharing aspects also.

# Appendix A

# Query Operator Evaluation Schemes

The schemes used by the query optimizer for evaluating the database operators are discussed below. All the schemes for an operator use different amount of memory and have different cost. The memory usage of the schemes excludes the input buffers and the output buffer. Cost of a scheme is the expected time of computation. Relations $R$ and $S$ are the right and left inputs to the operators with $m$ and $n$ tuples respectively. $R$ is the input to the unary operators. The following notations are used:

$p$: size of a pointer
$L$: average length of join attribute of $R$
$comp$: time to compare the join keys.
$b$: number of hash buckets. determined by the hash function.
$f$: hash table fudge factor.
$hash$: time to hash a key.
$mwrite$: time to write to main memory.
$mread$: time to read from main memory.
$fwrite$: time to write to flash memory.
$fread$: time to read from flash memory.

### A.0.1 Selection and Projection

The schemes for Selection and Projection operators are shown in Figure A.0.1.

### A.0.2 Join

In a left-deep tree join strategy, the right input $R$ is always a base relation and the left input $S$ is pipelined. Hence, only nested loop join and hash join are considered. The following are the join schemes and their costs and memory usage.

| Projection Cost |
|---|
| $m * fread$ |

| Selection Cost |
|---|
| $m * (fread + comp)$ |

Figure A.1: Selection and Projection Cost

**Nested Loop Join(NLJ)**

Nested loop join does not require any temporary memory structures except for the two tuple buffers.

| Scheme | Cost | Reason |
|---|---|---|
| NLJ | $(n * fread) + (m * n) *$ $(fread + comp)$ | Read and compare in nested loop manner |

| Scheme | Memory | Reason |
|---|---|---|
| NLJ | 0 | No extra memory needed |

Figure A.2: NLJ

**Indexed Nested Loop Join(BSTINLJ)**

In this scheme, an in-memory search tree index is created on the right input relation $R$ and the index is probed for joining a tuple from $S$. Binary search trees are considered as they are simpler.

| Scheme | Cost | Reason |
|---|---|---|
| BSTINLJ | $m * (fread$ $+ mwrite)$ $+ \sum_{i=1}^{m}(comp *$ $\log i)$ $+ n * (comp * \log m$ $+ fread)$ $+ (n * fread)$ | Reading the tuple writing the index Lookup during index creation Lookup into the index |

| Scheme | Memory | Reason |
|---|---|---|
| BSTINLJ | $(3p + L) * m$ | Two pointers for left and right child |

Figure A.3: BSTINLJ

The term for lookup during creation $\sum_{i=1}^{m}(comp * \log i)$ can be approximated to $(comp * m \log m)$.

### Hash Join(HJ)

In this scheme, a hash table is created for the right input relation $R$. The hash table contains pointers to the tuples and not the key values. Since the key values are not stored in hash table, the number of lookup values to check will be $\frac{m}{b}$. When a tuple from $S$ arrives, it is hashed, the hash value is used to lookup into the hash table to obtain the candidate tuples from $R$ for joining.

| Scheme | Cost | Reason |
|---|---|---|
| HJ | $m*(fread+$ $hash+mwrite)$ $+n*fread+n*hash$ $+n*(\frac{m}{b})*$ $(comp+fread+mread)$ | Reading the tuple writing the index hashing into the table traversing the hash bucket |
| Scheme | Memory | Reason |
| HJ | $(2p*m)*f$ | Pointers to the tuple and to the next bucket element |

Figure A.4: HJ

### A.0.3 Aggregation

Nested loop scheme and buffered scheme are used for aggregation. For the following schemes the input is $R$. Some more parameters are

$d$: number of distinct groupby values.

$aggr$: time to do the aggregation for one tuple.

$val$: size of the aggregation value.

$sub$: cost of the subtree below the aggregate operator in the query plan

#### Nested Loop Aggregation(NLA)

In this scheme as many iterations over the input are performed as there are distinct groupby values, and one initial iteration to get the value to start with. At each iteration, the aggregation for the current value is done and the next value to be aggregated is determined. Only the current value to be aggregated and the next value is stored in main memory. Hence, memory requirement is nil. When the aggregate operator is *sort*, at each iteration the tuple is output instead of aggregating.

Note that cost and memory for NLAAvg(For average, we need to maintain a count) and NLASort are the same as NLASum.

#### Buffered Aggregation(BA)

In this scheme, there is only one iteration over iterate over the input. Main memory is required for storing the aggregates. The aggregates are constructed on the fly.

| Scheme | Cost | Reason |
|---|---|---|
| NLASum | $(1+d)*m*(fread+aggr)$ $+(d*sub)$ | Nested loop read and aggr |

| Scheme | Memory | Reason |
|---|---|---|
| NLASum | 0 | No additional memory |

Figure A.5: NLASUM

| Scheme | Cost | Reason |
|---|---|---|
| BASum | $m*(fread+aggr)$ | Reading the input only once |

| Scheme | Memory | Reason |
|---|---|---|
| BASum | $(d*val)$ | Memory to store all aggregate results |

Figure A.6: BASUM

The cost and memory for BAAvg is the same as BASum. There is no Buffered Aggregation scheme for *sort* operator since the size of the results is equal to the size of the relation.

# Appendix B

# Query Results

The performance graphs for Q1 are based on the query execution plan (QEP) shown in Figure B.1. Table B.1, Table B.2, Table B.3 and Table B.4 show the amount of RAM available, schemes chosen by the optimizer based on the available RAM and the estimated time required for the execution of the QEP with these schemes.

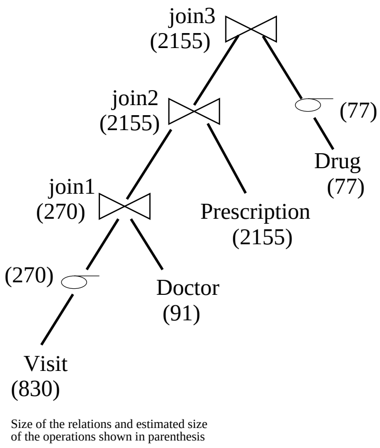



Figure B.1: QEP of Q1

The symbols NLJ, HASH and INLJ in the following tables represent nested loop join, hash join and index nested loop join respectively.

Table B.1: Performance of Q1 when compression is not used

| **RAM Size** (*bytes*) | **Scheme (join1, join2, join3)** | **Execution time** (*sec*) |
|---|---|---|
| 0 - 799 | NLJ NLJ NLJ | 7.68 - 12.89 |
| 800 - 1231 | NLJ NLJ HASH | 6.59 - 10.45 |
| 1232 - 2177 | NLJ NLJ INLJ | 6.35 - 10.22 |
| 2178 - 2687 | HASH NLJ INLJ | 6.20 - 10.11 |
| 2688 - 23642 | INLJ NLJ INLJ | 5.33 - 8.77 |
| 23643 - 24588 | NLJ HASH INLJ | 0.77 - 1.7 |
| 24589 - 25099 | HASH HASH INLJ | 0.53 - 1.3 |
| 25100 - 30000 | INLJ HASH INLJ | 0.42 - 0.6 |

Table B.2: Performance of Q1 with file level compression

| **RAM Size** (*bytes*) | **Scheme (join1, join2, join3)** | **Execution time** (*sec*) |
|---|---|---|
| 0 - 2647 | NLJ NLJ NLJ | 179.49 - 184.7 |
| 2648 - 2999 | NLJ NLJ HASH | 101.26 - 105.45 |
| 3000 - 4361 | NLJ NLJ INLJ | 101.02 - 105.22 |
| 4362 - 4871 | HASH NLJ INLJ | 89.5 - 93.37 |
| 4872 - 40882 | INLJ NLJ INLJ | 88.63 - 92.36 |
| 40883 - 41828 | NLJ HASH INLJ | 11.44 - 12.37 |
| 41829 - 42338 | HASH HASH INLJ | 1.39 - 2.32 |
| 42339 - 44999 | INLJ HASH INLJ | 1.15 - 1.92 |
| 45000 - 50000 | INLJ HASH INLJ | 1.04 - 1.22 |

Table B.3: Performance of Q1 with record level compression

| **RAM Size** (*bytes*) | **Scheme (join1, join2, join3)** | **Execution time** (*sec*) |
|---|---|---|
| 0 - 799 | NLJ NLJ NLJ | 8.45 - 13.66 |
| 800 - 1231 | NLJ NLJ HASH | 7.2 - 11.06 |
| 1232 - 2177 | NLJ NLJ INLJ | 6.95 - 10.82 |
| 2178 - 2687 | HASH NLJ INLJ | 6.79 - 10.7 |
| 2688 - 23642 | INLJ NLJ INLJ | 5.91 - 9.35 |
| 23643 - 24588 | NLJ HASH INLJ | 0.81 - 1.74 |
| 24589 - 25099 | HASH HASH INLJ | 0.55 - 1.32 |
| 25100 - 30000 | INLJ HASH INLJ | 0.44 - 0.62 |

Table B.4: Performance of Q1 with attribute level compression

| **RAM Size** (*bytes*) | **Scheme (join1, join2, join3)** | **Execution time** (*sec*) |
|---|---|---|
| 0 - 799 | NLJ NLJ NLJ | 7.68 - 12.89 |
| 800 - 1000 | NLJ NLJ HASH | 6.59 - 10.45 |
| 1001 - 1946 | NLJ NLJ INLJ | 6.35 - 10.22 |
| 1947 - 2183 | HASH NLJ INLJ | 6.20 - 10.11 |
| 2184 - 22411 | INLJ NLJ INLJ | 5.33 - 8.77 |
| 22412 - 23412 | NLJ HASH INLJ | 0.77 - 1.7 |
| 23413 - 24358 | HASH HASH INLJ | 0.53 - 1.3 |
| 24359 - 24595 | INLJ HASH INLJ | 0.42 - 0.6 |

# Bibliography


[Act] Microsoft Activesync. microsoft.com/windowsmobile/about/syncup.mspx.

[AGP98] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. One-Phase Commit : Does It Make Sense? In *ICPADS*, 1998.

[Aqu] AQUA. http://www.mlasia.iitb.ac.in/aaqua.htm.

[BBPV00] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling down Database Techniques for the Smartcard. In *VLDB*, 2000.

[BK02] Stephen Blott and Henry F. Korth. An Almost Serial Protocol for Transaction Execution in Main Memory Database Systems. In *VLDB*, 2002.

[BPA00] Christophe Bobineau, Philippe Pucheral, and Maha Abdallah. A Unilateral Commit Protocol for Mobile and Disconnected Computing. In *ICPADS*, 2000.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization in Compressed Database Systems. In *ACM SIGMOD*, 2001.

[Che02] Z. Chen. *Building Compressed Database Systems*. PhD thesis, Cornell University, 2002.

[DB2] DB2 Everyplace. http://www-3.ibm.com/software/data/db2/everyplace.

[Ddh] `http://www.ddhsoftware.com`.

[DG00] R.A. Dirckze and Le Gruenwald. A Pre-Serialization Transaction Management Technique for Mobile Multidatabases. In *Mobile Networks and Applications*, 2000.

[DHB97] M.H. Dunham, A. Helal, and S. Balakrishnan. A Mobile Transaction Model that Captures both the Data and Movement Behaviour. In *Mobile Networks and Applications*, 1997.

[DVR02] A. Datta, D. VanderMeer, and K. Ramamritham. Parallel Star Join + DataIndexes: Efficient Query Processing in Data Warehouses and OLAP. In *TKDE*, 2002.

[FAB94] M. Farach, A. Amir, and G. Benson. Let Sleeping Files Lie: Pattern Matching in Z-compressed Files. In *SODA*, 1994.

[Gig01] E. Giguere. Mobile Data Managemnt: Challenges of Wireless and Offline Data Access. In *ICDE*, 2001.

[GRS98] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *ICDE*, 1998.

[GS91] G. Graefe and L. Shapiro. Data Compression and Database Performance. In *ACM/IEEE-CS Symp. on Applied Computing, pages 22-27*, 1991.

[Ham] B. Hammond. Merge Replication in Microsoft's SQL Server 7.0.

[HRS95] J.R. Haritsa, G. Ray, and S. Seshadri. Database Compression: A Performance Enhancement Tool. In *COMAD*, 1995.

[HSS00] A. Hulgeri, S. Sudarshan, and S. Seshadri. Memory Cognizant Query Optimization. In *COMAD*, 2000.

[HWKM00] S. Helmer, T. Westmann, D. Kossmann, and G. Moerkatte. The Implementation and Performance of Compressed Databases. In *ACM SIGMOD*, 2000.

[KD97] J.S. Keen and W.J. Dally. Extended Ephemeral Logging: Log Storage Management for Appliactions with Long Lived Transactions. In *ACM TODS*, 1997.

[KLLP01] J. Karlsson, A. Lal, C. Leung, and T. Pham. IBM DB2 Everyplace: A Small Footprint Relational Database System. In *ICDE*, 2001.

[Lan90] Rom Langerak. View Updates in Relational Databases with an Independent Scheme. In *ACM PODS*, 1990.

[OLi] Oracle Lite. http://otn.oracle.com/products/lite/content.html.

[Ray95] G. Ray. Data Compression in Databases. Master's thesis, IISc Bangalore, 1995.

[SC90] J. Stamos and F. Cristian. A Low-cost Atomic Commit Protocol. In *IEEE Symposium on Reliable Distributed Systems*, 1990.

[SC93] J. Stamos and F. Cristian. Coordinator Log Transaction Execution Protocol. In *Distributed and Parallel Databases*, 1993.

[Sen04] R. Sen. An Open Source DBMS for Handheld devices. Master's thesis, IIT Bombay, 2004.

[SG00] P. Sheshadri and P. Garrett. SQL Server for Windows CE-A Database Engine for Mobile and Embedded Platforms. In *ICDE*, 2000.

[Sim] The Simputer. http://www.simputer.org.

[SKS02] A. Silberscatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. 2002.

[Sol00] D. Solomon. *Data Compression: The Complete Reference*. Springer-Verlag Inc., New York, 2000.

[TTP+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H.Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *ACM Symposium on Operating System Principles*, 1995.

[WC99] G.D. Walborn and P.K. Chrysanthis. Transaction Processing in PROMOTION. In *ACM SAC*, 1999.