A Survey of Physical Design Techniques of Information Systems -TEST

Karim Ali and Sarah Nadi
{karim, snadi}@cs.uwaterloo.ca
David R. Cheriton School of Computer Science
University of Waterloo

June 30, 2010

Abstract

The abstract goes here.

1 Introduction

Database users write queries and updates using a "user-friendly" language such as SQL without having to worry about how the underlying data is stored. Physical database design is concerned with the actual storage of data on disk. This means how the files are organized, and how they are accessed. Data is stored in the form of records in files which reside on a physical disk. In this sense, we try to find the most efficient way possible to store and access the data such that queries are answered and executed efficiently. This is reflected in designing proper indexing techniques, partitioning data for more efficient access, designing materialized views, data clustering, data compression, striping, mirroring and denormalization [14].

This survey presents the different physical design patterns that have been used in different types of information systems. The systems discussed in this survey are disk based relational databases, main memory relational databases, data warehouses, and XML databases. The aim of this survey is not to explain the details of every data structure used in physical design, but rather to compare how the different data structures have been used in these systems. This includes explaining how alterations or additions are made to data structures to fit the specific needs of every system. For a complete reference on the details of the data structures used in physical database design, please see (CITE HERE).

The rest of the survey is organized as follows. Section 2 first explains different elements of physical database design. For example, this includes explaining what indexes are and briefly mentioning what the different types of indexes used in information systems are. After explaining the different elements of physical database design, Section 3 discusses how each of the four systems uses these elements to improve its efficiency. Sections 3.1, 3.2, 3.3, 3.4 discuss Disk Based Databases, Main Memory Databases, Data

Warehouses, and XML Databases respectively. Throughout these sections, comparisons are made as to how the different elements have been modified to fit the needs of each system. Section 5 then presents physical database design can be automated. Section 6 mentions some of the open problems of physical database design, and suggests possible solutions. Finally Section 7 summarizes and concludes this summary.

2 Elements of Physical Database Design

2.1 Indexes

The first physical database design decision to be considered is the choice of indexes to be implemented. The concept of indices has been around for a long time. Just like and index at the end of the book is provided to help find certain content faster, indices in an information system help find content in tables faster. The main idea of an index is to have each record in a table have a unique identifier (primary key) and then organize these identifiers in a certain way that allows for fast access of a specific record. Of course, indexing can be done on any column in the database and not necessarily the primary key. There are many type of indexes that have been used in information systems.

B-trees (short for Balanced Trees) are the most used database index structure. The B-tree is essentially similar to the traditional Binary Search Tree, but instead of having one value per node, a B-tee can have many values per node citecomer1979ubiquitous as shown in Figure (PUT A FIGURE FOR A B TREE HERE). B-trees are suitable for relational databases since the cost of retrieval in a B-tree is at most proportional to: $log_d \frac{n+1}{2}$. The cost of insertion and deletion is at most proportional to $log_d n$ due to the possibility of progressing back up the tree to balance it after an insertion or a deletion. Although B-trees do well in retrieval, deletion and insertion, they do not perform well in sequential search.

Different variations of B-trees have been used. For example, $B^+ - trees$ are now the main method of indexing in Disk Based Relational Databases (DRDB). These will be discussed in Section 3.1. The T Tree has been proposed for main memory databases, and will be discussed in Section 3.2. Cache sensitive indexes were then proposed. A variation of T Trees, Cache Sensitive T-Trees (CST-Trees) was introduced, and is also discussed in Section 3.2. On the same note, Cache Conscious $B^+ - trees$ and Prefetching $J^+ - Trees$ are also discussed in Section 3.2.

Of course, there are other indexing techniques used such as hash tables, and bitmap indices. Bitmap indices are used in Data Warehouses, and will be discussed in details in Section 3.3.

2.2 Materialized Views

In a large database system, there are usually a few complex queries that require the joining of many large tables. If these queries are frequently run, then the database performance is likely to suffer, and users have to wait for a long time to get the results back. If we know these queries beforehand, it makes sense to simply store the results of the queries on disk instead of recomputing them each time. This is exactly what materialized views aim to do. By precalculating the results of a complex query, and storing them in a

table on disk, the new table with the results is very likely to be much smaller than the original tables decreasing I/O access costs and thus increasing performance [14]. Once a materialized view exists, the user can either explicitly query the materialized view, or enter the original query, and have the query optimizer rewrite the query to use the materialized views instead of the original tables. Query rewriting using materialized views is a very popular database research topic (CITE SOME REFERENCES), but is beyond the scope of this survey.

Despite the fact that materialized views provide a significant improvement in performance, we cannot simply create a materialized view for each common query. To begin with, materialized views consume disk storage which might be limited. The second, and main, problem with materialized views is their maintenance. Ensuring that the views have the most up to date data is tricky, and may outset the benefit of using materialized views if it is not properly designed. Finally, having several materialized views may increase the cost of searching for the appropriate view to use during the query optimization stage. Data warehouses heavily rely on materialized views since they have many queries with several joins. This will be discussed in details in Section 3.3.

Many query optimization techniques have relied on rewriting queries using views (E.g. [13, 9, 8, 1]).

2.3 Partitioning

Partitioning is an important aspect of physical design as it reduces table scanning time. There are two main categories of partitioning: horizontal partitioning and vertical partitioning. Horizontal partitioning divides the table into sets of table rows where each row or record still has all the attributes of the table. For example, dividing the data by date where all data dating less than year 2000 lies in one partition while all data dating more lies in another. On the other hand, vertical partitioning reduces the width of the dataset by storing some attributes in one table, and some in another.

The main advantage of any type of partitioning is to reduce the amount of time it takes to scan a table which in turn improves performance. For example, if a table has 1000 records, and a query is only using the first 20 records then having these 20 records in a partition containing 50 records will save time as opposed to examining all the 1000 records.

2.4 Clustering

2.5 Data Compression

2.6 Other Techniques

Striping, mirroring, denormalization.

3 Physical Design of Different Information Systems

Given the different physical design methods explained in the previous section, we now look at how these methods apply to different types of information systems. Mainly, how

they apply to Disk-based relational database systems, main memory relational database systems, data warehouses, and XML databases.

3.1 Disk Based Relational Database System (DRDB)

Relational Databases were first introduced by Codd [5] In relational databases, users specify the actions they want to execute without having to worry about how these operations will actually be performed. This is where the role of physical design comes in. Physical design specifies the access paths used to get data from the tables. This includes which indices to use, and in which order to access the tables [6] and so on.

There are many variations of B-trees. In relational databases, the most popular index is the $B^+ - tree$. The $B^+ - tree$ is a variant of the B tree which is becoming the main indexing method supported by many databases such as DB2, Oracle, and SQL Server citelightstone2007physical. The main difference between the $B^+ - trees$ and the B-tree is that only leaf nodes contain data pointers in a $B^+ - tree$. (COMPARE COST HERE). In Prefix $B^+ - trees$, instead of using the actual key value of a record, a prefix from the key value is used to decrease the storage size needed which will in turn allow more records to be stored per node thus decreasing the height of the tree.

The B+tree is the main indexing method used in current relational databases [14]. B+trees have been successful in relational databases since data is ultimately recorded in files. Since B+trees have a high fanout, they allow less I/O operations to access a specific data record which is stored in a specific file.

3.2 Main Memory Relational Database System (MMDB)

A Main Memory Database (MMDB) is one where the data resides in the main memory of the system rather than on a disk [7]. It should be noted that this is different from caching. Disk based database systems use the main memory to cache query results. However, MMDB's primary copy of the data resides in main memory. MMDB have many implementation challenges that have been addressed throughout the research community.

Garcia-Molina and Salem [7] mention some of the challenges involved. One is concurrency control. In disk based databases, locks are tracked through a hash table, but the actual objects on disk do not contain lock information. On the other hand, in MMDB, lock status is part of the object itself since it is cheap to keep a number of bits for that. The next challenge is access methods. In disk-based databases, B-Trees are used to index the data for faster access. B-Trees have a short bushy nature to try to decrease the height of the tree to decrease the number of I/O accesses. Since I/O access is not a problem in MMDB, longer tree structures are used since its cheap to access main memory. Another point is that data values do not need to be stored in the index itself since they will be stored in main memory anyways so there are no performance gains from storing them in the index itself.

3.2.1 Type of Indexes Used

Two considerations are taken when designing index structures for MMDB. The first is that the data resides in main memory and not on disk, and so many of the considerations

taken for I/O operations is no longer there. The second is having index structures cache conscious. Lots of research work has shown that cache performance is very critical in MMDBs [19, 3]. This is because the big difference between processor speed and main memory access speed. Thus, a cache miss is still relatively expensive in main memory databases. Accordingly, we will see that many of the index data structures proposed for main memory databases focused on being cache conscious or cache sensitive.

$B^+ - Trees$

The $B^+ - Tree$, explained in the previous section, is originally designed for disk based database systems. However, it has also been found useful for main memory databases (REF) as it is cache conscious.

T Trees

Lehmen and Carey [12] study the different features needed for indices of MMDB. When designing data structures for MMDB, we are mainly concerned with the efficient use of CPU cycles and memory. Accordingly, the main goal of an index structure for MMDB is to minimize the overall computation time while using the minimum memory possible. Since all the database data is in main memory, index structures do not need to store actual values like those in disk-oriented databases. Instead, a pointer to the data instead of the actual data can be stored. Lehmen and Carey introduce a new index structure, the T Tree, which combines the features from AVL Trees and B Trees that are suited to main memory. Figure 1 shows the proposed structure of the T Tree. They run some experiments on data residing in main memory to compare the performance of the T Tree to existing structures. All structures were modified to contain pointers to data values rather than the data values themselves. The structures included were AVL Trees, simple arrays, B Trees, Chained Bucket Hashing, Extendible Hashing, Linear Hashing and Modified Linear Hashing. Their experiments showed that for unordered data, Modified Linear Hashing gave the best performance, and for ordered data, T Trees gave the best performance for a mix of searches, inserts and deletes.

Cache Sensitive Search Trees (CSS-Trees)

[19].

Cache Sensitive T-Trees (CST-Trees)

T Trees were the main index structure used for main memory databases for some time after their proposal. However, it was discovered that B+-trees outperform T-Trees on modern processors because of the growth of cache miss latency relative to processor speed [19, 11]. This is because the height of the tree is high which makes the total number of memory accesses from the root to the leaf node higher. The other reason is that the node size is not aligned with the cache line size. The other problem with T-Trees is that there is a big waste of space in the cache because unnecessary data is brought into the cache. Additionally, record pointers stored in the tree take up a lot of space.

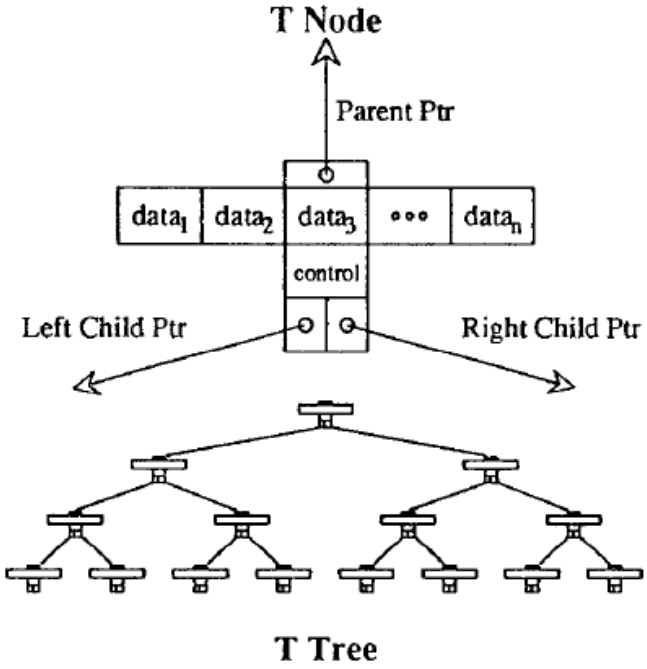

Figure 1: T Tree by Lehmen and Carey [12]

To resolve these issues, Lee et al. citelee2007cst propose the Cache Sensitive T-Trees (CST-Trees). This is achieved in the following ways. First, a binary search tree whose node values are the maximum key of each node in the T Tre is constructed. That way, for a given value, the node containing this value could be quickly located through searching the binary tree. Second, the need to store pointers is eliminated by storing the tree in an array and locating the necessary nodes through index calculation. Finally, node sizes are aligned with cache line size such that there are no cache misses when accessing data in each binary search tree in the array.

Cache Sensitive $B^+ - trees$ **(**$CSB^+ - trees$**)**

Rao and Ross [19] show that $B^+ - trees$ are more cache conscious than binary search trees and T-Trees which means they are more suited to main memory databases. However, traditional B+-Trees still had poor cache performance. Therefore, they propose Cache Sensitive B+-Trees ($CSB^+ - Trees$) [20] as an index structure for main memory. To do so, they eliminated most of the child pointers and had more keys in each node to improve locality and reduce tree height [15]. Since the number of cache misses in search operations is proportional to the height of the tree, $CSB^+ - trees$ have fewer cache misses, and thus better performance.

$Prefetching B^+ - Trees (pB^+ - trees)$

Chen et al. [4] propose using prefetching to improve the performance of $B^+ - trees$. Since current computer systems can prefetch different data simultaneously, the author take advantage of this. The node size of the $B^+ - tree$ is made wider than the cache line

size. This way, several cache misses can be overlapped at the same time thus improving performance.

$J^+ - trees$ **and** $pJ^+ - trees$

3.3 Data Warehouses

3.4 XML Databases

XML (eXtensible Markup Language) is increasingly being used as a data exchange format. Accordingly, the ability to store and manage XML documents is needed. Salminen and Tompa [21] define an XML database as "a collection of XML documents and their parts, maintained by a system having capabilities to manage and control the collection itself and the information represented by that collection." Salminena and Tompa also highlight some of the requirements of an XML database system. These requirements include dealing with namespaces, Internet resources, querying parts of an XML document, and transformations of XML documents.

There are two ways to store XML documents in a database. The first is to have the database internally translate the XML document into relational tables (XML-enabled databases). The second is to have data structures that can persistently store XML documents (native XML databases). eXist [16] and TIMBER [10] are two of the famous research efforts to design a native XML database.

3.4.1 Index Structures

Despite the special nature of XML documents, some of the index structures used in relational databases can still be utilized in XML databases. However, some adjustments may need to be made to fit the nature of the XML documents. For XML-enabled databases, XML documents are usually represented as relational tables, and then indexed similar to other tables. Pal et al. [18] describe how this is done in Microsoft SQL Server 2005. To start with, a new data type called 'XML' was introduced. This data type could contain values of complete XML documents, or just fragments of XML data. First, the different nodes in the XML data are labelled with the ORDPATH [17] mechanism. The XML data is then shredded into a relational table with five main columns:ORDPATH, TAG, NODE_TYPE, VALUE, and PATH_ID. This information is then stored in a $B^+ - tree$ as the primary index of the XML type. Additionally, a secondary index can be created on any of the columns of the primary index to speed things up.

For native XML databases such as eXist [16], $B^+ - trees$ were still used to index the XML documents. However, in contrast to XML-enabled databases, the XML data is not first stripped into a relational table. eXist uses a number schema to assign unique identifiers to each node in the XML document. These unique number identifiers allow the determination of any node's parent, sibling or possible child nodes. Four index files are created for XML data where all the indexes are based on $B^+ - trees$. One index manages the collection hierarchy, one index collects nodes in a paged file and associates unique node identifiers to the actual nodes, one index indexes elements and attributes, and the last index keeps track of word occurrences and is used by the fulltext search extensions.

3.4.2 Materialized Views

Since XPath query processing is expensive, materialized views come in handy in XML databases. Balmin et al. [2] propose a framework for using XPath materialized views for XML query processing. (TALK ABOUT IT MORE)

4 Real Databases (incorporate in previous section)

Oracle Database uses B tree indexes and their subtypes such as Index-organized tables, reverse key indexes, descending indexes, B-tree cluster indexes. It also uses Bitmap and bitmap join indexes, function based indexes and domain based indexes.

MySQL mostly uses B trees, except for spatial data where R trees are used and hash indexes are used for memory tables.

5 Automating Physical Database Design

6 Open Problems in Physical Database Design

Indexing in the cloud [22].

7 Conclusions

Acknowledgment

The authors would like to thank...

References

[1] S. Abiteboul and O.M. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, page 263. ACM, 1998.

[2] A. Balmin, F.
"Ozcan, K.S. Beyer, R.J. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, page 71. VLDB Endowment, 2004.

[3] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 54–65. Citeseer, 1999.

[4] S. Chen, P.B. Gibbons, and T.C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, page 246. ACM, 2001.

[5] EF Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):387, 1970.

[6] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems (TODS)*, 13(1):128, 1988.

[7] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, pages 509–516, 1992.

[8] J. Goldstein and P.Å. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, page 342. ACM, 2001.

[9] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the International Conference on Very Large Data Bases*, pages 358–369. Citeseer, 1995.

[10] HV Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, et al. Timber: A native XML database. *The VLDB journal*, 11(4):274–291, 2002.

[11] I. Lee, J. Shim, S. Lee, and J. Chun. CST-trees: cache sensitive t-trees. In *Proceedings of the 12th international conference on Database systems for advanced applications*, pages 398–409. Springer-Verlag, 2007.

[12] T.J. Lehman and M.J. Carey. A study of index structures for main memory database management systems. In *Conference on Very Large Data Bases*, volume 294, 1986.

[13] A.Y. Levy, A.O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM, 1995.

[14] S. Lightstone, T.J. Teorey, and T. Nadeau. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more.* Morgan Kaufmann, 2007.

[15] H. Luan, X.Y. Du, and S. Wang. Prefetching $J^+ - Tree$: A Cache-Optimized Main Memory Database Index Structure. *Journal of Computer Science and Technology*, 24(4):687–707, 2009.

[16] W. Meier. eXist: An open source native XML database. *Web, Web-Services, and Database Systems*, pages 169–183, 2003.

[17] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: insert-friendly XML node labels. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908. ACM, 2004.

[18] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML data stored in a relational database. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, page 1157. VLDB Endowment, 2004.

[19] J. Rao and K.A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, page 89. Morgan Kaufmann Publishers Inc., 1999.

[20] J. Rao and K.A. Ross. Making B+-trees cache conscious in main memory. *ACM SIGMOD Record*, 29(2):475–486, 2000.

[21] A. Salminen and F.W. Tompa. Requirements for XML document database systems. In *Proceedings of the 2001 ACM Symposium on Document engineering*, page 94. ACM, 2001.

[22] J. Wang, S. Wu, H. Gao, J. Li, and B.C. Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 international conference on Management of data*, pages 591–602. ACM, 2010.