A Survey of Physical Design Techniques of Information Systems

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

July 10, 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 [28].

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 [11] 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. For example, Bitmap indices are commonly 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. Oracle 8i (SOURCE) introduced materialized views to fit this purpose. 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 [28, 6]. 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. Many query optimization techniques have relied on rewriting queries using views (E.g. [27, 19, 16, 1]), but discussing the details of the rewriting 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.

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. Further classification in partitioning is single vs group horizontal or vertical partitioning [18]. Horizontal partitioning just divides the data into partitions where each group contains complete tuples. Group horizontal partitioning groups tuples that are more frequently used together. Single vertical partitioning divides a relation into groups each of which contains only one type of attribute. Group vertical partitioning divides the data vertically such that attributes that are commonly required together are physically stored together.

2.4 Clustering

When only records satisfying certain criteria are need from a table, it is often unnecessary to scan the whole table. For example, if only records that occurred in January 2010 are needed, then querying all the table would be an extremely unnecessary overhead. This is where clustering comes in. Clustering reflects how the records are physically located together on disk. Records that are likely to be queried together will be stored physically together to avoid querying the whole table. For the previously mentioned example, the data can be clustered by month, such that all the records of January 2010 are physically

in the same page, for example. In this case, only this page will need to be read. As people realized the potential of clustering, multidimensional clustering was introduced where data is clustered based on different criteria at the same time.

2.5 Other Physical Design Techniques

There are other physical design methods to improve the performance of the information systems [28]. For example, data compression can be used to make more data fit into a fixed amount of disk space such that it can be access faster. Data striping is used to distribute data that is accessed together across multiple disks. This allows the data to be retrieved faster through parallelism. Database reliability is also improved through mirorring which involves duplicating the data on multiple disks. Finally, refining the global schema to reflect query and transaction requirements is also sometimes used. This is called 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. We mainly focus on the first four techniques discussed in Section 2 as these are the main design decisions involved in physical design.

3.1 Disk Based Relational Database System (DRDB)

Relational Databases were first introduced by Codd [10] 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 [13] 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 [28]. 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) (sometimes referred to as in-memory database) is one where the data resides in the main memory of the system rather than on a disk [15]. 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 [15] 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 Index Structures

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. Instead of focusing on disk access and disk storage, the data structures for main memory databases should focus on the efficient use of CPU cycles and memory [26]. The second is having index structures cache conscious. Cache memories improve performance by holding recently referenced data [41]. If the memory reference is found in the cache, then execution proceeds at processor speed. Otherwise, the data has to be fetched from main memory. With the big difference between processor speed and main memory speed, cache misses are still relatively expensive. Lots of research work has shown that cache performance is very critical in MMDBs [5, 36]. 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. In summary, a main memory index should be able to reduce overall computation time while using as little memory as possible [26]. It should also be noted that unlike disk based systems where it is advantageous to store actual attribute values in the index, main memory systems do not require that. Instead, pointers to the actual attribute values can be used. This, in turn, will reduce the size of the index which is preferable in this case.

In this section, we talk about three families of index structures used in MMDB: B Trees, T Trees, and Binary Search Trees. There are more recent index structures developed for MMDB such as J+ Trees [29], but which have not gained much popularity, and so we do not discuss them in our survey. Table 1 summarizes the features of the different index structures discussed in this section with respect to MMDB.

	Features suitable for MMDB	Cache Consciousness
B Trees	• Good storage utilization • Reasonably quick searching • Fast updating	• Reasonable cache behavior if node fits in cache line
B+ Trees	• Wastes space since all data is stored in the leaves • Reasonably quick searching • Fast updating	• Reasonable cache behavior if node fits in cache line
CSB+ Trees	• Same features of B+ Trees	• Good cache performance due to improved locality & lower tree height
pB+ Trees	• Same features of B+ Trees	• Improves cache miss performance by prefetching more data in each cache miss • Node size is also increased leading to the same benefits of CSB+ Trees
T Tree	• Contain pointers to data values instead of the values themselves leading to better space utilization • Node size is bigger leading to previous mentioned advantages	• Cache behavior was not considered at time of design
CST	• Fast traversal in $log_{m+1}n$	• Good cache behavior as number of key nodes per line is chosen according to cache line miss
CSS Trees	• Fast traversal in $log_{m+1}n$ (m is the number of keys per node) • Mainly suitable for read environments	• Good cache behavior since m is chosen to fit in the cache line size

Table 1: Summary of index structures for Main Memory Databases (MMDB)

B Trees

Although B Trees, explained in the previous section, are originally designed for disk based database systems, they were found to also be useful in MMDB [26]. In disk based systems, the B+ Tree is more used than the regular B Tree. However, in main memory databases,

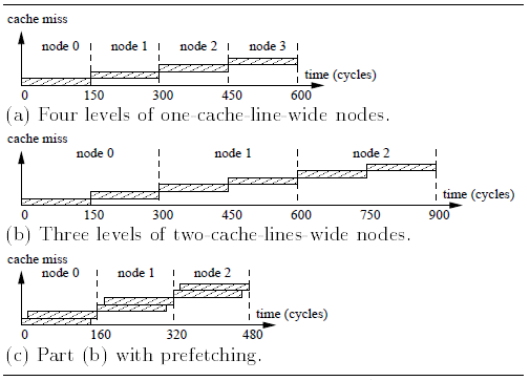

Figure 1: Performance of various B+ Tree searches where a cache miss to memory takes 150 cycles and a subsequent access can begin 10 cycles later [8]

the B Tree would be more suitable from a storage perspective since there is no gain from keeping all the data in the leaves. In a main memory system, this would only waste space without improving performance. On the other hand, the B+ Tree uses multiple keys to search within a node. This means that if the node fits in a cache line, this cache load can satisfy more than more comparison leading to better cache utilization [36]. Therefore, B+ Trees in general have reasonable cache behavior.

However, the cache behavior of B+ Trees could still be improved. Therefore, Rao and Ross [37] propose Cache Sensitive B+-Trees ($CSB^+ - Trees$) [37] 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 [29]. 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.

Chen et al. [8] propose using prefetching to improve the performance of B+ Trees. They call the new index structure prefetching B+ Tree (pB+ Tree). Since current computer systems can prefetch different data simultaneously, the authors take advantage of this. The idea here is to increase the node size of B+ Trees and employ prefectching at the same time. In a B+ Tree, every time we move down a level in the tree, a cache miss occurs. Accordingly, prefetching larger sizes nodes means that we can overlap multiple cache misses before they occur. This is because in one cache miss, extra data is retrieved in parallel with the originally requested data if it can be predicted sufficiently early. Figure 1 shows how prefetching retrieves the same data in less time. This method, however, does not eliminate the full cache miss latency with every level in the tree.

T Trees

Lehmen and Carey [26] 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 2 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 compared included AVL Trees, simple arrays, B Trees, Chained Bucket Hashing, Extendible Hashing, Linear Hashing and Modified Linear Hash-

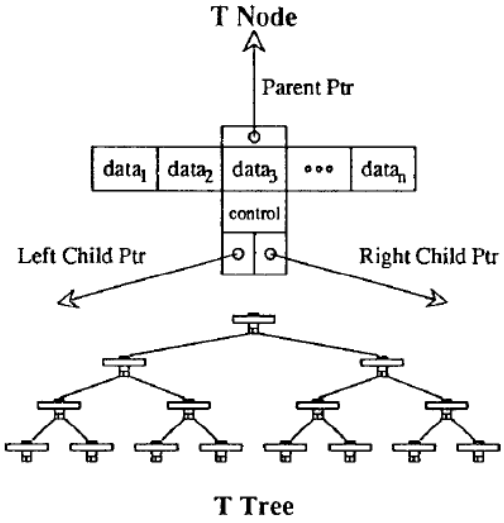

Figure 2: T Tree by Lehmen and Carey [26]

ing. 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. This is mainly because a T node contains many elements which results in good update and storage characteristics. The T Tree was the first index structure specifically designed for main memory databases.

Although T Trees have been specifically designed for MMDB, they did not consider the cache behavior at that time [36]. Although many keys are fit into one node, only the two end keys are used for comparison leading to low node utilization. Similarly, array binary search trees have the same problem.

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 [36, 25]. 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. Another 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.

To resolve these issues, Lee et al. [25] 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 Tree 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 Search Trees (CSS-Trees)

Binary Search Trees have poor cache behavior when the array is much bigger than the cache. Rao and Ross [36] propose the Cache Sensitive Search Tree (CSS-Tree) to improve this cache behavior. To do that, the search tree is stored as a directory in an array. The number of keys per node is chosen such that the whole node fits in a cache line. This improves local searching within a node to just one cache miss. They also hard-code the traversal within a node such that finding the next node happens in $log_{m+1}n$ times rather than log_2n times in a binary search tree where m is the number of keys per node. However, CSS Trees are only suitable for read environments since they eliminate nearly all child pointers, thus removing the support for incremental update.

3.2.2 Materialized Views

From our literature survey, it seems that materialized views are not used in MMDB. This makes sense since the problems materialized views try to address are not present in MMDB. That is, the cost of computing complicated join queries is much less in MMDB due to the low cost of accessing the data in memory. Accordingly, the maintenance cost of materialized views in MMDB will outweigh their minimal benefit.

3.2.3 Partitioning

Partitioning is used in disk based systems to divide the data into the physical pages in a way that will yield the best performance. This could be through vertical partitioning or horizontal partitioning, as previously explained. In main memory databases, we do not have the issue of expensive disk access which makes partitioning in memory unnecessary as it will not save any costs. However, partitioning is needed in the secondary storage of the data on disk [18]. Disk storage is still used as a backup of main memory databases. When a crash occurs, a reload of the database from disk or archive memory takes place. In order to avoid page faults caused by referencing data that is still being reloaded, the reloading process must be designed to be efficient in loading the important data such that the database users are not affected. An efficient MMDB reload takes into consideration the number of I/Os for reload and the number of main memory references during transaction processing. Gruenwald and Eich [18, 17] show that horizontal partitioning is the most suitable when the database performs more modifications than tuple deletions and more selections that projections and joins. Single vertical partitioning is chosen in the other cases. They also show that if we just concentrate on the reload performance (i.e the number of I/Os for reload), then single vertical partitioning is always the best choice.

3.2.4 Clustering

Clustering in disk based systems is important because sequential I/O access is cheaper than random or dispersed I/O access. This is not the case with main memory, and so clustering is not needed in MMDB [15, 31]. Therefore, components of one object may be spread across memory without impacting performance.

3.3 Data Warehouses

A data warehouse is a subject-oriented, integrated, time-variant, and non-volatile collection of data and decision support technologies, aimed at enabling the knowledge worker (executive, manager, analyst) to make better and faster decisions [21, 7]. More than half of IT executives named Data warehousing as the highest-priority post-millennium project for them [40]. The value of data warehousing for an organization depends on the organization's need for reliable, consolidated, unique and integrated reporting and analysis of its data, at different levels of aggregation.

Data warehousing has been shown useful in many industries: manufacturing (for order shipment and customer support), retail (for user profiling and inventory management), financial services (for claims analysis, risk analysis, credit card analysis, and fraud detection), transportation (for fleet management), telecommunications (for call analysis and fraud detection), utilities (for power usage analysis), and healthcare (for outcomes analysis) [7].

Data warehouses tend to be extremely large, in fact it is quite possible for a data warehouse to store tens of petabytes of data while loading tens of terabytes of data everyday [44]. Datta et al. [12] notes that the information in a data warehouse is usually multidimensional in nature, requiring the capability to view the data from a variety of perspectives. Aggregated and summarized data become more crucial than detailed records in such environment. Therefore, the workloads are query intensive with mostly ad hoc, complex queries, often requiring computationally expensive operations such as scans, joins, and aggregation. Performing such operations on large amounts of data, like in the case of data warehousing, complicates the situation further. Moreover, the results have to be delivered interactively to the business analyst using the system.

Although data in a data warehouse is extracted and loaded from multiple on-line transaction processing (OLTP) data sources (including DB2, Oracle, IMS databases, and flat files) using Extract, Transfer, and Load (ETL) tools (see Figure 3), a data warehouse is usually maintained seperately from the organization's operational databases [40, 7]. This architecture can be justified owing to the fact that operational databases are finely tuned to satisfy known OLTP requirements and functionalities which are quite different from that of on-line analytical processing (OLAP) which is supported by data warehousing. Analyzing data for decision support usually requires consolidating data from many heterogeneous sources of varying quality, or use inconsistent representations, codes and formats. Moreover, understanding trends or making predictions requires historical data, whereas operational databases store only current data. In OLAP, there's a need to support multidimensional data models and operations which requires special data organization, access methods, and implementation methods, not generally provided by DBMSs targeted for OLTP [7]. Finally, in data warehousing, query throughput and response times are more important than transaction throughput which is the major performance metric for operational databases.

star schema fact table

Typical Operations * Data Cleaning * Load * Refresh

Joins in data warehouses are usually expensive since the fact table (the largest one) participates in every join [12]

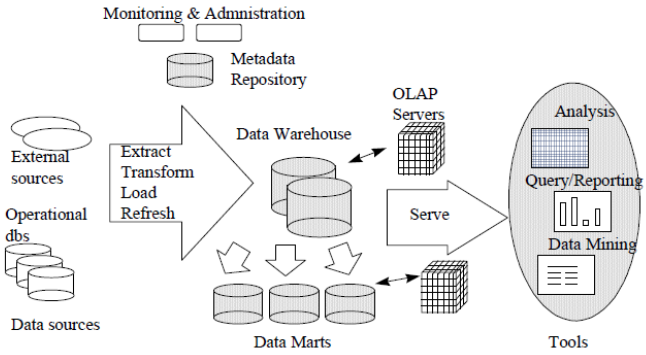

Figure 3: Data Warehousing Architecture [7]

3.3.1 Index Structures

Tree-based Indexes B-tree (B+ trees but commonly referred to as B-trees) [34], R-tree [20, 9], K-D-B-tree [38], BV-tree [14], UB-tree [4]

Bitmapped Join Indexes [33]

Projection Indexes Involves positional indexing where tuples are accessed based on their ordinal position [12]

Bit-sliced Indexes

3.3.2 Partitioning

[44]

Vertical Partitioning

Horizontal Partitioning

3.3.3 Materialized Views

not very useful since most queries are ad-hoc [2] [43]

3.3.4 Data Cubes

[9] ROLAP o Data is stored in relational databases. o Support extensions to SQL. o Efficiently implement the multidimensional data model and operations. o Ex: Microstrategy. MOLAP o Directly store multidimensional data in special data structures (arrays). o Implement the OLAP operations over these special data structures. o Star Schema. o Snowflake Schema. [24] o Fact Constellations. o Ex: Essbase (Arbor), statistical DBMSes.

hierarchical data clusters [23]

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 [39] 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 in which an XML database system can be designed. 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). In this section, we will discuss how the database's physical design needs to be adapted in both cases. Other challenges such as expressing and executing queries and updates are faced in XML database design, but these are beyond the scope of this survey.

3.4.1 Index Structures

Despite the special nature of XML documents, the same 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. [35] 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 [32] 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 [30] or TIMBER [22], $B^+ - trees$ 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.

TIMBER also uses a similar numbering schema. There are minor differences between the two numbering techniques, but the main idea is the same. The numbering is based on three values of a node: its start label, its end label, and its level (i.e. nested depth). On the other hand, Sedna [42] uses a different reasoning to produce its numbering schema. Sedna assignes string identifiers to nodes such that they are lexically ordered according

to the position of the node. Despite the differences in the ways the numbering is constructed, it seems that that no novel indexing structure is needed for XML databases. The problem is rather how to assign the nodes unique identifiers (the numbers or labels in the numbering schema) that can be used as identifiers in the index to reflect the XML document structure.

3.4.2 Materialized Views

Since XPath query processing is expensive, materialized views come in handy in XML databases. Balmin et al. [3] 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 [45].

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] Robert Armstrong. Data warehousing: Dealing with the growing pains. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 199–205, Washington, DC, USA, 1997. IEEE Computer Society.

[3] A. Balmin, F. Özcan, 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.

[4] R. Bayer. The universal B-tree for multidimensional indexing: General concepts. *Worldwide Computing and Its Applications*, pages 198–209, 1997.

[5] 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.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[7] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[8] 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.

[9] D.W. Cheung, B. Zhou, B. Kao, H. Kan, and S.D. Lee. Towards the building of a dense-region-based OLAP system. *Data & Knowledge Engineering*, 36(1):1–27, 2001.

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

[11] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[12] Anindya Datta, Debra VanderMeer, and Krithi Ramamritham. Parallel star join + dataindexes: Efficient query processing in data warehouses and olap. *IEEE Trans. on Knowl. and Data Eng.*, 14(6):1299–1316, 2002.

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

[14] Michael Freeston. A general solution of the n-dimensional b-tree problem. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 80–91, New York, NY, USA, 1995. ACM.

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

[16] 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.

[17] L. Gruenwald and MH Eich. Choosing the best storage technique for a main memory database system. In *Information Technology, 1990.'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No. 90TH0326-9)*, pages 1–10, 1990.

[18] L. Gruenwald and MH Eich. Database partitioning techniques to support reload in a main memory database system: MARS. In *International Conference on Databases, Parallel Architectures and Their Applications,. PARBASE-90*, pages 107–109, 1990.

[19] 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.

[20] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM.

[21] W. H. Inmon. *Building the Data Warehouse,3rd Edition.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[22] 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.

[23] N. Karayannidis and T. Sellis. Hierarchical clustering for OLAP: the CUBE File approach. *The VLDB JournalThe International Journal on Very Large Data Bases*, 17(4):655, 2008.

[24] R. Kimball, M. Ross, W. Thorthwaite, B. Becker, and J. Mundy. *The data warehouse lifecycle toolkit.* Wiley-India, 2009.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] S.M. Moldovan. Databases: towards performance and scalability. 2008.

[32] 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.

[33] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, 1995.

[34] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49, New York, NY, USA, 1997. ACM.

[35] 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.

[36] 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.

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

[38] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, New York, NY, USA, 1981. ACM.

[39] 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.

[40] A. Sen and A.P. Sinha. A comparison of data warehousing methodologies. *Communications of the ACM*, 48(3):84, 2005.

[41] A.J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):530, 1982.

[42] I. Taranov, I. Shcheklein, A. Kalinin, L. Novak, S. Kuznetsov, R. Pastukhov, A. Boldakov, D. Turdakov, K. Antipin, A. Fomichev, et al. Sedna: native XML database management system (internals overview). In *Proceedings of the 2010 international conference on Management of data*, pages 1037–1046. ACM, 2010.

[43] Dimitri Theodoratos and Mokrane Bouzeghoub. A general framework for the view selection problem for data warehouse design and evolution. In *DOLAP '00: Proceedings of the 3rd ACM international workshop on Data warehousing and OLAP*, pages 1–8, New York, NY, USA, 2000. ACM.

[44] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 international conference on Management of data*, pages 1013–1020. ACM, 2010.

[45] 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.