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

Abstract

The physical design decisions taken during database design highly affect the database performance. These design decisions include the index structures used, which views can be materialized, how to partition the data etc. These decisions vary by the nature of the system being implemented. For example, the index structures used for disk-based database systems differ from those used in a memory-based database system. In this survey, we present the elements of physical design in different information systems. We start by describing the physical design of traditional disk-based relational database systems then compare it to other types of information systems. Specifically, we look at the physical design of Main Memory Database Systems (MMDB), XML Databases, and Data Warehouses. We examine how the specific nature of each of these systems affect its physical design.

1 Introduction

Relational Database Management Systems (RDBMS) have been the most popular type of Information Systems for decades. When we think of designing a database system, we usually think of which data we need to keep track of and how will we properly organize it into meaningful tables. However, this is not the only phase in database design. All of the designed tables need to be physically stored on disk. How these tables will be stored and accessed greatly affects the database performance. 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. In the traditional relational database system, this involved selecting proper indexing techniques, partitioning the data for more efficient access, materializing the results of some queries, clustering and compressing the data, and many other techniques [1, 2]. The physical design aspects of relational databases have been studied in much detail. However, over time, different types of Information Systems have emerged that have different characteristics. For example, most relational databases have been disk-based where the primary copy of the data resides on disk. This is as opposed to Main Memory Relational Databases (MMDB) where the

primary copy of the data resides in main memory and no disk operations are needed to query the data. Such differences pose new requirements for the physical design of these systems.

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 (traditional database systems), 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. Using the different physical design methods used in disk based RDBMSs, we explore how these methods apply to the other systems and where the differences and similarities lie.

The rest of the survey is organized as follows. Section 2 first explains different elements of physical database design used in traditional disk-based databases. Section 3 then discusses how each of the three other systems uses these elements, as applicable. Sections 3.1, 3.2, 3.3, discuss Main Memory Databases, XML Databases, and Data Warehouses respectively. Throughout these sections, comparisons are made as to how the different elements have been modified to fit the needs of each system. Section 4 mentions some of the open problems of physical database design, and suggests possible solutions. Finally, Section 5 summarizes and concludes this survey.

2 Elements of Physical Database Design

Relational Databases were first introduced by Codd [3]. Disk based Relational Database Management Systems (RDBMS) are the most common and popular type of Information Systems. Since they have been extensively researched and have been around for many decades, we use them as the basis of our comparison. In this section, we describe the different elements of physical design in RDBMS. These elements will also be examined in the other systems to see how they apply to them.

2.1 Index Structures

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.

Btrees (short for Balanced Trees) are the most used database index structure. The Btree 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 [4]. 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 current disk-based Relational Databases [2] such as DB2, Oracle, and SQL Server. The main difference between the B+tree and the Btree is that only leaf nodes contain data pointers in a B+tree, and leaf nodes contain pointers to each other as shown in Figure 1. 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. 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.

Bitmap indices represent another type of index structures that is commonly used and cited in the literature. Using bitmaps as indexes first appeared in the literature as part of the architecture of the Model 204 commercial database of Computer Corporation of America for the IBM 370 computer [5]. A bitmap index structure is based on a bitmap (i.e. bitset or bit array) data structure where individual bits correspond to values in unique records. Given a table column C where a record x can have one of six unique values, a traditional bitmap index defined for C generates one bitmap vector for each of the six different values ($B0 - B5$). If the value of C for record x is set to 3, the corresponding bit in vector $B3$ is set to 1. Otherwise the bit is set to 0.

Bitmap indices are best used when the set of possible key values in the index is small with a large number of rows, e.g. gender attribute [6]. Therefore, bitmap indices outperform B+trees for low cardinality attributes. The reason is that given an attribute with low cardinality, defining a B+tree index on it means that the tree will have very few leaf nodes, each one storing a very long list of record identifiers (RIDs). On the other hand, defining a bitmap index on that same attribute means that such long lists or RIDs will not be stored, rather bit vectors that represent the distinct attribute values will be stored as previously explained. This leads to a direct benefit of consuming less storage space. Moreover, queries in that case can be answered using bit-wise logical operations (e.g. and, or, not) which can improve query performance [**?**]. Another advantage of bitmap indices is the ability to perform more efficient joins by creating bitmap join indices and using those indices to calculate the result of the join [**?**]. Although there are many benefits gained from using bitmap indices, they poorly perform during updates. The reason is that when an updates is to be done to a bitmap index, the whole bit vector for the index is locked, blocking any access to the index for other operations, then the lock is released when the update is done. That is why bitmap indices are usually used for read-mostly databases, a famous example of which are data warehouses (Section 3.3).

Another variation of indexing, a hash table index, often simply known as a hash index is used to organize unordered physical database tables for efficient access [2]. This type of index is usually defined for keys that have unique values in the data records. Creating this index requires passing the attribute (key) value to a hash function that maps it to a starting block address (i.e. bucket address). Afterwards, table insertions will be done according to the hash function, as well as querying the records later.

In general, indices are a way of organizing the logical structure of a database without

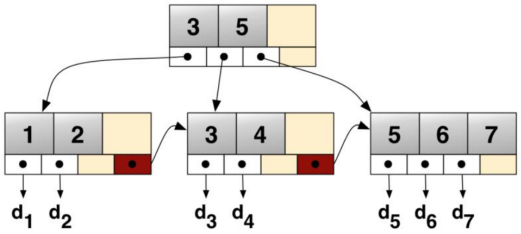

Figure 1: B+tree

altering its physical structure. However, sometimes it is desirable to store related records near each other on disk for better locality. This can be achieved through implementing a clustered index on the desired key values. Otherwise, the created index is known as non-clustered index. The primary advantage of a clustered index is therefore the ordering of the physical data rows in accordance with the index blocks that point to them which. Therefore, fewer data block reads are required to answer a given query. The only caveat with using a clustered index is that there can only be one clustered index per table because the physical organization of the table must be fixed [2].

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 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 [2, 7]. 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. [8, 9, 10, 11]), but discussing the details of the rewriting process 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 frequent query. First, 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.

2.3 Partitioning

Partitioning in physical database design is a method for reducing the workload on any one hardware component, like an individual disk, by partitioning (dividing) the data over several disks [2]. There are two main categories of partitioning: horizontal partitioning

	Range Partitioning	List Partitioning	Hash Partitioning	Composite Partitioning
How?	If key falls in range	List of keys	Hash function	Hybrid
Why?	Disk limits	Group point values	Range/List are N/A	According to needs
Benefits	Partition elimination Improved administration Fast roll-in/out Implicit clustering			
Example	- Zip code - Time range	- Regions - States	- EmployeeId - ProductId	Hash then List
RDBMS	MySQL, DB2, Oracle, PostgreSQL, SQL Server	MySQL, Oracle, PostgreSQL, SQL Server	MySQL, Oracle	MySQL, Oracle

Table 1: Types of horizontal partitioning

and vertical partitioning [12]. Horizontal partitioning divides the table into sets of table rows where each row or record still has all the attributes of the table [13, 14]. 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. Usually, less frequently queried columns are partitioned together to reduce the size of the table, especially in data warehouses where not all the columns of a table are analyzed for decision support.

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 [2]. 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. Horizontal partitioning is more widely used, and is implemented in all current DBMSes [15, 16, 17, 18, 19, 20] which explains why there exists many flavors of horizontal partitioning techniques. There are three major types of horizontal partitioning: range partitioning [21], list partitioning [2] (which is a variation of range partitioning), and hash partitioning [2]. Of course, one can mix and match those partitioning schemes and come up with a composite partitioning that fits a particular situation [2]. Table 1 shows different categories of horizontal partitioning, and the situations they are suited to. Combining partitioning and indexing can often lead to better performance. For example, if List partitioning is used to divide a table by regions and then a bitmap index is used on that table, finding records in the same region or state can be much faster. Finally, partitioning the data into different physical devices to achieve parallelism when accessing it can greatly increase the performance of the database.

2.4 Clustering

Clustering is the process of grouping related data together on disk for efficiency of access and maximizing resource utilization. Data clustering benefits from the fact that sequential I/O access in physical disks is much cheaper than random access. Therefore, records that are likely to be queried together are clustered (stored physically together) to decrease the number of random I/O accesses. Whenever the number of I/O random accesses is decreased, CPU blocks are also reduced which result in an overall reduction of CPU costs improving the query performance, especially for multidimensional queries (e.g. group by) [2]. The same concept of data clustering can similarly be applied to multiple dimensions

(columns or column attributes) of a table. Kennedy et al. [22] summarizes the benefits of multidimensional clustering as follows:

- Dramatic reduction in I/O requirements due to clustering.
- Scans on any dimension index provide clustered data access.
- Dimensions/slices can be accessed independently using their block indexes.
- Block index scans can be combined (ANDed, ORed).
- Access to the clustered data is much faster.

On the other hand, defining clustering keys, clustering scheme, and the granularity of clustering which makes the problem of designing an optimal clustering in this context has combinatorial complexity, having an extremely large search space [23].

2.5 Other Physical Design Techniques

There are other physical design methods to improve the performance of the information systems [2]. 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. However, compression is only beneficial if the cost of executing the compression algorithm is lower than the I/O cost of accessing the data. Otherwise, compressing the data must just add an overhead without improving performance. Data striping is used to distribute data that is accessed together across multiple disks. This allows the data to be retrieved faster through parallelism. Striping has the same goals as partitioning. However, striping is more geared towards the hardware level which is beyond the control of the database while partitioning is within the control of the database. Database reliability is also improved through mirroring 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 which can be thought of as manual clustering of the data to improve performance.

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 Main Memory Relational Database Systems (MMDB), 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 Main Memory Relational Database System (MMDB)

A Main Memory Database (MMDB) (sometimes referred to as in-memory database) is one where the primary copy of the data resides in the main memory of the system rather than on a disk [24]. 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. Accessing data in main memory is faster than disk, and so MMDBs usually have better performance that disk based systems. However, main memory is more expensive and the size is limited in comparison to disk. However, MMDBs are essential to use where some of the collected data cannot, or will not, be stored on disk in the first place. For example, phone switch data may be needed for reasoning with other stored data, but it is captured in real time and will not necessarily be stored on disk. MMDB have many implementation challenges that have been addressed throughout the research community. These challenges will be discussed throughout this section.

3.1.1 Index Structures

Garcia-Molina and Salem [24] mention data access methods as one of the main challenges involved in MMDBs. 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 [25]. The second is having index structures cache conscious. Cache memories improve performance by holding recently referenced data [26]. 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 [27, 28]. 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 [25]. 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 2 summarizes the features of the different index structures discussed in this section with respect to MMDB. The T Tree is the most used index for MMDB (E.g. MySQL CLuster). However, there are other indexes used as well. For example, IBM's solidDB uses Trie (or prefix tree) for its indexing. For their purposes, it served as a good index for main memory since it eliminates the need for many key comparisons [30].

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 [25]. In disk based systems,

Index Structure	Features	Cache Consciousness
B Trees	-Good storage utilization -Reasonably quick searching -Fast updating	-Reasonable cache behavior if node fits in cache line
B+ Trees	-Reasonably quick searching -Fast updating	-Reasonable cache behavior if node fits in cache line
CSB+ Trees	-Same features of B+ Trees -Stores child nodes in an array. Stores pointer to first child only and uses index calculation to get the rest	-Good cache performance due to improved locality & lower tree height (because of larger node size)
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 -Many sorted keys per node -Only minimum and maximum keys used for comparisons -Contains record pointers in every key -Good update & storage characteristics if Btrees.	-Cache behavior was not considered at time of design -Record pointers waste space in cache
CST	-Binary search tree for maximum value of each node leads to faster search -More space efficient by removing pointers & using array storage. Index calculation used to allocate values	-Node size is aligned with cache line size to avoid misses -Higher usage of cached data in binary search tree
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 2: Summary of index structures for Main Memory Databases (MMDB)

the B+ Tree is more used than the regular B Tree. However, in main memory databases, the B Tree would be more suitable from a storage perspective since there is no gain from

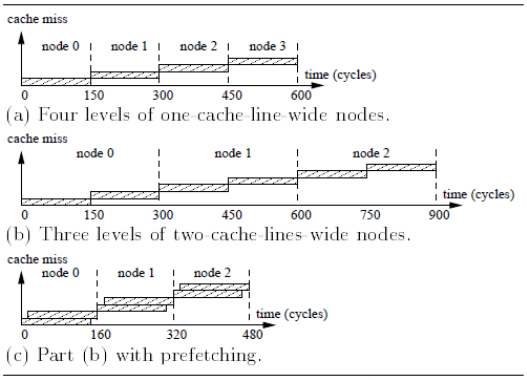

Figure 2: Performance of various B+ Tree searches where a cache miss to memory takes 150 cycles and a subsequent access can begin 10 cycles later [32]

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 [28]. Therefore, B+ Trees in general have reasonable cache behavior, and variations of them have been adapted in MMDBs.

However, the cache behavior of B+ Trees could still be improved. Therefore, Rao and Ross [31] propose the Cache Sensitive B+-Trees ($CSB^+ - Trees$) [31] as an index structure for main memory. To do so, they eliminated most of the child pointers by storing the child nodes in an array, and only keeping a pointed to the first child. Additionally, they used 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. [32] 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 2 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 [25] 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 3 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,

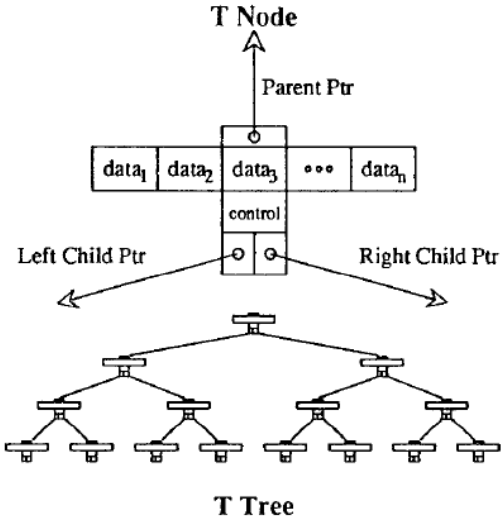

Figure 3: T Tree by Lehmen and Carey [25]

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. 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. The T Tree has many sorted keys per node, and only uses the minimum and maximum keys for comparison. Additionally, every key contains a record pointer.

Although T Trees have been specifically designed for MMDB, they did not consider the cache behavior at that time since the gap between processor speed and main memory speed was not big then [28]. 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. It was discovered that B+-trees outperform T-Trees on modern processors because of the growth of cache miss latency relative to processor speed [28, 33]. 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 which is not used (the record pointers) brought into the cache.

To resolve these issues, Lee et al. [33] propose the Cache Sensitive T-Trees (CST-Trees). 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. This produces a better utilization of the cache by removing the unused data. 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 [28] 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.

3.1.2 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. However, this does not mean that they cannot be used at all in MMDBs. As with all other physical design decisions, it rather depend on the benefits of implementing the technique. Accordingly, if the data is very large and may benefit from using materialized views, there is no reason not to use them.

3.1.3 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 [34]. 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 [34, 35] show that horizontal partitioning is the most suitable when the database performs more modifications than tuple deletions and more selections that projections and joins. 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 vertical partitioning is always the best choice.

3.1.4 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 [24, 36]. Therefore, components of one object may be spread across memory without impacting performance.

3.2 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 [37] 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 querying parts of an XML document, and transformations of XML documents. The main challenge in XML databases is path navigation since it is a very expensive operation.

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). XML-enabled databases are suited for data-centric documents which have a regular structure without any mixed content [38]. On the other hand, for document-centric documents which have an irregular structure with lots of mixed content, having a native XML database is essential. Examples of XML-enabled databases include SQL Server 2005 and Oracle, and examples of native XML databases include Tamino [39], eXist [40], and TIMBER [41].

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.

3.2.1 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. [42] 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 labeled with the ORDPATH [43] 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 [40] or TIMBER [41], B+trees are 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 full text search extensions. Oracle uses a similar technique for indexing its XML content where an XMLIndex table is created for XML entries [44]. This table contains a unique identifier for the XPath leading to the node, the row id of the table used to store the XML data,an order key that identifies the hierarchal positioning of the node, a locater key used for fragment extraction, and the text value of the node.

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 [45] uses a different reasoning to produce its numbering schema. Sedna assigns 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 existing indexing techniques can be adapted for XML databases. The challenge is 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. Additionally, in native XML databases, documents are usually grouped into collections, and an index may be used to search within these collections. This is different from the node index which searches within a document.

Besides indexing the actual nodes, path indexing is also common in XML databases since usually queries search for a specific path which can be very expensive. This is done by clustering together all the IDs of nodes on a given path, and creating an index for them [46, 47]. Other techniques include that proposed by Cooper et al. [48] where they propose an indexing mechanism for paths called "Index Fabric". An Index Fabric is based on Patricia tries [49] which are used for string indexing, but is customized for disk-based systems. A path in an XML document is encoded into a unique string, and is then inserted into the fabric. The encoded string is usually very short, and so searching is relatively fast.

3.2.2 Materialized Views

The main idea behind using materialized views is to rewrite queries using these views so that they run more efficiently. This is very important in XML Databases since XQuery and XPath queries are complicated [50]. Accordingly, materialized views are used in XML databases to improve performance. The semi-structured nature of XML data, and the expressiveness of most XML querying languages, view selection for optimization becomes a more complicated problem [51]. Therefore, the choice of which views to materialize becomes more complicated in XML database. Much work has been done on rewriting XML queries using views [50, 52, 53, 54]. However, this problem is not within the scope this survey.

3.2.3 Partitioning

In XML-enabled databases, inlining during the shredding of the data such that children nodes that occur only once are stored with the parent node produces better partitioned

data [55]. Inlining shreds a document into sets of tables according to the document schema which leads to better performance because queries tend to access less data. Such inlining is similar to horizontal partitioning [56]. Additionally, it is common to horizontally partition the mapped relational tables such that nodes of the same type are stored together [57].

3.2.4 Clustering

Clustering in XML databases follows the heuristic that sub-elements are likely to be queries with an element [41]. Accordingly, elements and their subelements are clustered together. In general, storing the XML data in document order it thought to be the best technique for efficient querying. Additionally, Lian et al. [58] suggest clustering XML documents by structural similarity to produce more efficient access. This is done by representing the documents as a structure graph (s-graph), and then identifying clusters based on similarity using a clustering algorithm. Nayak [59] also proposes a similar clustering mechanism for XML data. Her technique is also based on structural similarity, but it is more focused on the level structure of the XML documents.

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 [60, 61]. More than half of IT executives named Data warehousing as the highest-priority post-millennium project for them [62]. 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) [61].

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 [63]. Datta et al. [64] 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 (Figure 4), a data warehouse is usually maintained seperately from the organization's operational databases [62, 61].

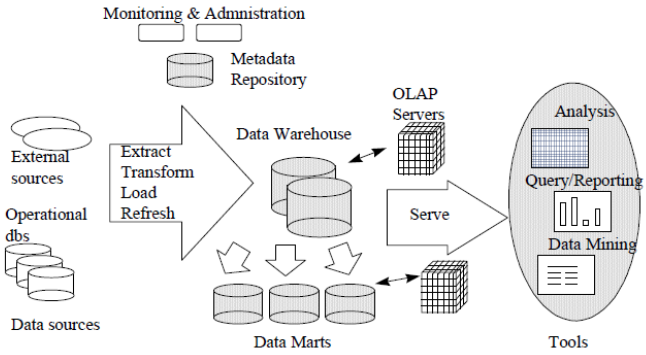

Figure 4: Data Warehousing Architecture [61]

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 [61]. Finally, in data warehousing, query throughput and response times are more important than transaction throughput which is the major performance metric for operational databases.

The design of a data warehouse can follow one of two paradigms: relational-based OLAP (ROLAP) or multidimensional OLAP (MOLAP). In ROLAP, data is stored in relational databases and the support for multidimensional data is achieved through the use of the star schema [65]. A star schema (Figure 5) is a database schema where a fact table T that holds the base data in the database. Other tables in the database are dimensions for table T, where each dimension table is linked to the fact table through foreign keys in T pointing to the primary keys in the dimension tables. Thus, the fact table, which is always the largest table in such schema, participates in almost all join operations [64]. This might enforce some design decisions for which indices to use, whether materialized views should be implemented to reduce the overhead of such expensive join operation or not. In MOLAP, multidimensional data is directly stored in special data structures (arrays) forming data cubes where each side of the cube reflect a dimension of the base data. OLAP operations are implemented on top of these special data structures. Usually a snowflake schema [66] or fact constellations are used to logically design the data warehouse.

The rest of this section will discuss how such challenges and requirements of data warehouse design affect the selection of physical design elements during the design process.

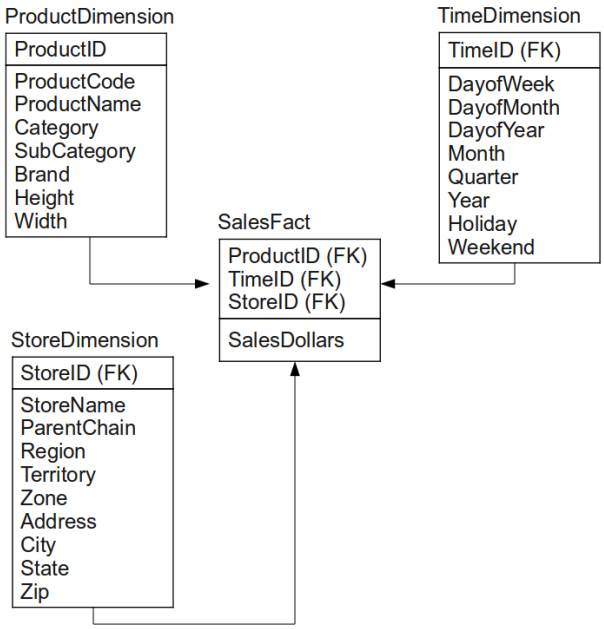

Figure 5: An example of a star schema

3.3.1 Index Structures

Despite the design challenges involved in data warehousing, some relational index structures are still used after adapting them to match those challenges.

Tree-based Indices The widely used B-trees have been modified to suit the requirements of a data warehouse design, such modification resulted in various tree-based indices: UB-trees, BV-trees, K-D-B-trees, and R-trees. A UB-tree index [67] is a variation of B+trees where support for multidimensional data is provided by linearizing such data to be able to store it in one dimension. Usually the Z-order (Morton-order) [68] is used to perform this type of linearization. UB-trees inherit the advantages of B+trees, so they are well suited for high cardinality attributes. They are also excellent for point and interval queries (using the pointer in each leaf node to the next leaf node for faster index traversal). UB-trees also inherit the disadvantages of B+trees. Therefore, they poorly perform for low cardinality attributes.

A BV-tree [69] is an unbalanced tree that maintains the characteristics of the B-tree in n dimensions, and it can be degenerated to a balanced tree in the one-dimensional case. A BV-tree index was an attempt to solve the problem of generalizing the characteristics of a B-tree index in n dimensions. However, this type of indices did not really catch up in the literature of data warehouses since it has a very grave disadvantage. The design of a BV-tree index suggests that the size of the index can span multiple-page for very large files (which is the case in data warehouses). Since searching a BV-tree index might involve backtracking [69], such property is not desirable.

K-D-B-trees [70] is another type of tree-based indices that tries to combine the multidimensional search efficiency of K-D-trees [71] and the I/O efficiency of B-trees [72].

Another widely used tree-based index is R-tree [73]. An R-tree is an extension of a B-

tree with index records in its leaf nodes containing pointers to data objects. Nodes in an R-tree correspond to disk pages if the index is disk-resident. The structure of an R-tree is designed so that a spatial search requires visiting only a small number of nodes. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required. R-trees support multidimensional spaces by storing an n-dimensional rectangle, which is the bounding box of the spatial object indexed, in the index record entries [73]. R-trees were extended by Norbert et al [74] by designing the R*-tree to support both sparse and dense data at the same time with a slightly higher cost (during insertion) than R-trees. Cheung et al. [65] describe how R-trees can be used to index dense data regions (MOLAP data cubes), while having pointers from the leaf nodes to sparse data points (ROLAP).

Bitmap Join Index Bitmap join indices are another widely used type of indices in data warehouses epecially when a star schema is used. A bitmap join index is simply a bitmap index that is used for calculating the product of the join of the dimension tables. The result is then joined with the fact table to answer the given query [6]. In fact such process is less costly than joining the fact table with the dimension tables right away. The reason is that the fact table is usually larger than the product of the dimension tables. The disadvantage of bitmap join being not amenable to index updates is not really applicable to the case of data warehousing since a data warehouse can be considered as a read-mostly database.

Projection Index Not all columns in a data warehouse are of interest to the given query workloads. Therefore, it makes sense to have faster access to the columns that are used the most to improve the query performance. That is why Projection indices are often used in data warehousing. A projection index involves positional indexing of a column (form of vertical partitioning) where tuples are accessed based on their ordinal position [64]. More results per I/O operation is a direct benefit of using projection indices, since only data of interest are accessed and retrieved. However, only raw data can be fetched (e.g. column list in a selection clause). A projection index can be further extended into a bit-sliced index [75] where the projection index is sliced vertically into columns of bits. Each column is a bit-sliced index. Bit-sliced indices can help reduce storage costs by using a number of bits per value which is not a power of 2. For example, a SALES column contains all sales made in the last month, in pennies. Assuming no item cost more than $2^{20} - 1$ pennies (= \$10485.75), this is 20 bit-slice indices). Moreover, computing aggregates, checking predicates can be done using efficient base-2 arithmetic operation. This greatly enhances query performance since aggregates are crucial for data warehouses.

3.3.2 Materialized Views

not very useful since most queries are ad-hoc [76, 77]

3.3.3 Partitioning

[63]

Vertical Partitioning

Horizontal Partitioning

4 Open Research Problems in Physical Database Design

Automating the physical database design choices has been a very attractive area of research that is still receiving lots of attention until now. Choosing which indexes to create, which views to materialize, and how to partition data are hard choices. They are highly dependent on the schema used as well as the query workload. The number of combinations of choices that can be made is very large making this task quite tedious and difficult for database administrators. Accordingly, researchers have been inclined to find ways to automate these decisions. There were many early research efforts done in this area(E.g. [1, 78, 79, 61, 80, 81]), but most of these efforts focused on having a static workload. Accordingly, research shifted to more dynamic ways to automate physical design. Initial attempts used an dynamic, on-line approaches(E.g. [82, 83, 84]) where they observed the changes in the workload and adjusted the design accordingly while attempting to predict future behavior based on the observed behavior. To avoid the problems of on-line design, researchers are also currently looking at how to perform this dynamic physical design off-line (E.g. [85, 86]). Additionally, finding better evaluation techniques to compare different automatic physical design decision is currently being studied [87].

Other problems which are currently being researched include improving I/O operations in database systems by using Solid State Drives (SSD) instead of of Hard Disk Drives (HDD) [88, 89, 90]. SSDs have faster read performance, but perform poorly in random writes. However, for certain applications in databases and for databases where random writes are not as frequent, SSDs might be suitable. Using SDDs also seems promising when used for transaction logs, and intermediate results of sorting and joining [90].

Just like all other research areas are moving to the cloud, database systems are moving there too. Accordingly, lots of current research focuses on adapting storage techniques and index structures to the cloud. For example, Wang et al. propose a new index structure for indexing multi-dimensional data in the cloud [91]. Lots of work is currently being done in this area. For example, Wu et al. [92] propose a modified B+tree index for datasets stored in each compute node. The exact requirements and solutions for data management in the cloud are still being researched [93], and so this is an area that is expected to receive more attention in the future as cloud computing becomes more prevalent.

This is by no means an exhaustive list of the problems still being researched in physical database design, but they provide a flavor of the challenges still being faced. As shown, as new technologies such as SSDs or cloud computing environments emerge, database systems will need to adapt themselves to either make use of them or to be able to cope with the new requirements they pose.

	Index Structures	Materialized Views	Partitioning	Clustering
RDBMS	E.g. B+trees, Bitmap Index, Hash Tree	Store results of common complicated queries on disk	Horizontal Partitioning (range, list, hash, composite) Vertical Partitioning	Group related items together on disk
MMDB	-No need to store actual data values in index -Larger node size that is aligned with cache line size -E.g. B+tree, CSB+tree, T tree, CST tree	Not highly utilized since processing and main memory access is cheap	Only needed in secondary storage to speed up reloading in case of a crash -Horizontal & Vertical partitioning	Not needed since random access in main memory costs the same as sequential access
XML DB	Need a number scheme. E.g. B+tree and variations used.	XQuery and XPath results are materialized	Horizontal partitioning based on inlining or node type	Based on document order or structural similarity
Data Warehouses	E.g. Universal B+tree, Bitmap index, Projection Index	-Essential to improve performance -Not applicable to adhoc queries	-Horizontal partitioning based on time dimension -Vertical partitioning of seldom used columns.	Data is clustered by nature.

Table 3: Summary of the different elements of physical design in the studied systems

5 Conclusions

In this survey, we present four elements of physical design in database: index structures, materialized views, partitioning, and clustering. We started out by explaining how these design factors are implemented in traditional Relational Database Management Systems (RDBMS), and then presented how they are used in other information systems. We focused on three types of information systems: Main Memory Database Systems (MMDB), XML Databases, and Data Warehouses. We presented the challenges involved with each type of system, and how each of the four physical design elements were adapted or utilized to overcome these challenges. Table 3 summarizes the results of our survey. From our literature review, we found that the B+tree is the most utilized index structure, and has been adapted and customized to fit the different needs of other systems. For example, it was made more cache conscious to fit MMDB needs, and was changed to apply to multidimensions to fit the Data Warehouses needs. Materialized views were found to be heavily utlitized in all the systems we presented, except in MMDBs. Partitioning was also used in all systems, but was only used for data reloading in MMDBs. Horizontal partitioning is much more common that vertical partitioning. Clustering is very important in all systems, except for MMDBs. Clustering is especially very important for Data Warehouses where huge amounts of data are stored.

It was interesting to see how the different nature of these systems affects the physical design. Throughout the survey, we have showed some of the commonalities between the different systems. Moreover, understanding how to meet these needs would help us design mash-ups of these different systems. For example, just like relational database systems shifted into main memory, we might have an XML main memory database system. In that case, the index structures fitted for XML data indexing will have to be adapted for main memory by making sure they are aligned with the cache line size, for example. Similarly, if XML data warehouses are developed, more elaborate path indexing techniques and

more clustering would be required.

References

[1] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 91–128, 1988.

[2] S. S. Lightstone, T. J. Teorey, and T. Nadeau, *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more (The Morgan Kaufmann Series in Data Management Systems)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[3] E. F. Codd, "A relational model of data for large shared data banks," pp. 5–15, 1988.

[4] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[5] E. Patrick, "Model 204 Architecture and Performance," in *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, 1987, pp. 40–59.

[6] P. O'Neil and G. Graefe, "Multi-table joins through bitmapped join indices," *SIGMOD Rec.*, vol. 24, no. 3, pp. 8–11, 1995.

[7] S. Chaudhuri, "An overview of query optimization in relational systems," in *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 1998, pp. 34–43.

[8] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv, "Answering queries using views (extended abstract)," in *PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 1995, pp. 95–104.

[9] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-query processing in data warehousing environments," in *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 358–369.

[10] J. Goldstein and P.-A. Larson, "Optimizing queries using materialized views: a practical, scalable solution," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2001, pp. 331–342.

[11] S. Abiteboul and O. M. Duschka, "Complexity of answering queries using materialized views," in *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 1998, pp. 254–263.

[12] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 359–370.

[13] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal data partitioning in database design," in *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. ACM, 1982, p. 136.

[14] D. Shin and K. Irani, "Partitioning a relational database horizontally using a knowledge-based approach," in *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*. ACM, 1985, pp. 95–105.

[15] "MySQL 5.5 Data Partitioning," http://dev.mysql.com/doc/refman/5.5/en/partitioning-types.html.

[16] "Microsfot SQL Server Data Partitioning," http://msdn.microsoft.com/en-us/library/ms190787.aspx.

[17] "Sybase Data Partitioning," http://www.sybase.com/detail?id=1036923.

[18] "PostgreSQL Data Partitioning," http://www.postgresql.org/docs/current/interactive/ddl-partitioning.html.

[19] "Oracle Data Partitioning," http://www.oracle.com/us/products/database/options/partitioning/index.htm.

[20] "DB2 Data Partitioning," http://www.ibm.com/developerworks/data/library/techarticle/dm-0605ahuja2/.

[21] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "GAMMA-A High Performance Dataflow Database Machine," in *Proceedings of the 12th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1986, pp. 228–237.

[22] J. P. Kennedy, "Introduction to mdc with db2 udb luw," IBM, Tech. Rep., 2005.

[23] H. Jagadish, L. Lakshmanan, and D. Srivastava, "Snakes and sandwiches: Optimal clustering strategies for a data warehouse," *ACM SIGMOD Record*, vol. 28, no. 2, p. 48, 1999.

[24] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, pp. 509–516, 1992.

[25] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 294–303.

[26] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.

[27] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 54–65.

[28] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 78–89.

[29] H. Luan, X. Du, and S. Wang, "Prefetching $J^+ - Tree$: A Cache-Optimized Main Memory Database Index Structure," *Journal of Computer Science and Technology*, vol. 24, no. 4, pp. 687–707, 2009.

[30] S. H. Anotoni Walski, "solidDB and the secrets of speed. How the IBM in-memory database redefines high performance," Tech. Rep., 2010.

[31] J. Rao and K. A. Ross, "Making b+- trees cache conscious in main memory," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2000, pp. 475–486.

[32] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2001, pp. 235–246.

[33] I.-h. Lee, J. Shim, S.-g. Lee, and J. Chun, "Cst-trees: cache sensitive t-trees," in *DASFAA'07: Proceedings of the 12th international conference on Database systems for advanced applications*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 398–409.

[34] L. Gruenwald and M. 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*, 1990, pp. 107–109.

[35] ——, "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)*, 1990, pp. 1–10.

[36] S. Moldovan, "Databases: towards performance and scalability," 2008.

[37] A. Salminen and F. W. Tompa, "Requirements for xml document database systems," in *DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering*. New York, NY, USA: ACM, 2001, pp. 85–94.

[38] R. Bourret, "XML and Databases," 2003.

[39] Software AG., "Tamino XML database," http://www.softwareag.com/tamino/.

[40] W. Meier, "exist: An open source native xml database," in *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*. London, UK: Springer-Verlag, 2003, pp. 169–183.

[41] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A native XML database," *The VLDB Journal*, vol. 11, no. 4, pp. 274–291, 2002.

[42] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov, "Indexing xml data stored in a relational database," in *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 2004, pp. 1146–1157.

[43] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "Ordpaths: insert-friendly xml node labels," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 903–908.

[44] D. Adams, "Oracle xml db developer's guide. 11g release 1 (11.1)," Oracle, Tech. Rep., 2008.

[45] I. Taranov, I. Shcheklein, A. Kalinin, L. Novak, S. Kuznetsov, R. Pastukhov, A. Boldakov, D. Turdakov, K. Antipin, A. Fomichev, P. Pleshachkov, P. Velikhov, N. Zavaritski, M. Grinev, M. Grineva, and D. Lizorkin, "Sedna: native xml database management system (internals overview)," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. New York, NY, USA: ACM, 2010, pp. 1037–1046.

[46] T. Milo and D. Suciu, "Index structures for path expressions," in *ICDT '99: Proceedings of the 7th International Conference on Database Theory*. London, UK: Springer-Verlag, 1999, pp. 277–295.

[47] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese, "Path summaries and path partitioning in modern XML databases," *World Wide Web*, vol. 11, no. 1, pp. 117–151, 2008.

[48] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A fast index for semistructured data," in *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 341–350.

[49] D. Knuth, *The Art of Computer Programming, Vol. III Sorting and Searching*. Addison Wesley, Reading, MA, 1998.

[50] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for xml queries," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 87–98.

[51] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. Boncz, "Materialized view selection in xml databases," in *DASFAA '09: Proceedings of the 14th International Conference on Database Systems for Advanced Applications*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 616–630.

[52] 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 *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 2004, pp. 60–71.

[53] K. Aouiche, P. Jouve, and J. Darmont, "Clustering-based materialized view selection in data warehouses," in *Advances in Databases and Information Systems*. Springer, 2006, pp. 81–95.

[54] N. Tang, J. X. Yu, M. T. Ozsu, B. Choi, and K.-F. Wong, "Multiple materialized view selection for xpath query rewriting," in *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 873–882.

[55] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and querying ordered xml using a relational database system," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2002, pp. 204–215.

[56] M. Ramanath, J. Freire, J. Haritsa, and P. Roy, "Searching for efficient XML-to-relational mappings," *Database And Xml Technologies*, pp. 19–36, 2003.

[57] S. Amer-Yahia and M. Fernandez, "Overview of existing XML storage techniques," *submitted for publication*, 2002.

[58] W. Lian, D. W. lok Cheung, N. Mamoulis, and S.-M. Yiu, "An efficient and scalable algorithm for clustering xml documents by structure," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, pp. 82–96, 2004.

[59] R. Nayak, "Fast and effective clustering of xml data using structural information," *Knowl. Inf. Syst.*, vol. 14, no. 2, pp. 197–215, 2008.

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

[61] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Rec.*, vol. 26, no. 1, pp. 65–74, 1997.

[62] A. Sen and A. P. Sinha, "A comparison of data warehousing methodologies," *Commun. ACM*, vol. 48, no. 3, pp. 79–84, 2005.

[63] 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*. ACM, 2010, pp. 1013–1020.

[64] A. Datta, D. VanderMeer, and K. Ramamritham, "Parallel star join + dataindexes: Efficient query processing in data warehouses and olap," *IEEE Trans. on Knowl. and Data Eng.*, vol. 14, no. 6, pp. 1299–1316, 2002.

[65] D. W. Cheung, B. Zhou, B. Kao, H. Kan, and S. D. Lee, "Towards the building of a dense-region-based olap system," *Data Knowl. Eng.*, vol. 36, no. 1, pp. 1–27, 2001.

[66] R. Kimball, M. Ross, W. Thornthwaite, J. Mundy, and B. Becker, *The Data Warehouse Lifecycle Toolkit*. Wiley Publishing, 2008.

[67] R. Bayer, "The universal b-tree for multidimensional indexing: general concepts," in *WWCA '97: Proceedings of the International Conference on Worldwide Computing and Its Applications*. London, UK: Springer-Verlag, 1997, pp. 198–209.

[68] G. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," *IBM, Ottawa, Canada*, 1966.

[69] M. 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*. New York, NY, USA: ACM, 1995, pp. 80–91.

[70] J. 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*. New York, NY, USA: ACM, 1981, pp. 10–18.

[71] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[72] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. New York, NY, USA: ACM, 1970, pp. 107–141.

[73] A. 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*. New York, NY, USA: ACM, 1984, pp. 47–57.

[74] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, no. 2, pp. 322–331, 1990.

[75] P. O'Neil and D. Quass, "Improved query performance with variant indexes," in *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1997, pp. 38–49.

[76] R. Armstrong, "Data warehousing: Dealing with the growing pains," in *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 199–205.

[77] D. Theodoratos and M. 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*. New York, NY, USA: ACM, 2000, pp. 1–8.

[78] M. Hammer and A. Chan, "Index selection in a self-adaptive data base management system," in *SIGMOD '76: Proceedings of the 1976 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1976, pp. 1–8.

[79] M. R. Frank, E. Omiecinski, and S. B. Navathe, "Adaptive and automated index selection in rdbms," in *EDBT '92: Proceedings of the 3rd International Conference on Extending Database Technology*. London, UK: Springer-Verlag, 1992, pp. 277–292.

[80] B. Schiefer and G. Valentin, "DB2 universal database performance tuning," *Bulletin of the Technical Committee on*, p. 12, 1999.

[81] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley, "Db2 advisor: An optimizer smart enough to recommend its own indexes," *Data Engineering, International Conference on*, vol. 0, p. 101, 2000.

[82] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007*, 2007, pp. 826–835.

[83] K.-U. Sattler, I. Geist, and E. Schallehn, "Quiet: continuous query-driven index tuning," in *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, 2003, pp. 1129–1132.

[84] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-line index selection for shifting workloads," in *ICDEW '07: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 459–468.

[85] H. Voigt, W. Lehner, and K. Salem, "Poster session: Constrained dynamic physical database design," in *ICDEW '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering Workshop*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 63–70.

[86] S. Agrawal, E. Chu, and V. Narasayya, "Automatic physical design tuning: workload as a sequence," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 683–694.

[87] K. E. Gebaly and A. Aboulnaga, "Robustness in automatic physical database design," in *EDBT '08: Proceedings of the 11th international conference on Extending database technology*. New York, NY, USA: ACM, 2008, pp. 145–156.

[88] S.-W. Lee, B. Moon, and C. Park, "Advances in flash memory ssd technology for enterprise database applications," in *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2009, pp. 863–870.

[89] M. Du, Y. Zhao, and J. Le, "Using flash memory as storage for read-intensive database," *Database Technology and Applications, International Workshop on*, vol. 0, pp. 472–475, 2009.

[90] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1075–1086.

[91] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*. New York, NY, USA: ACM, 2010, pp. 591–602.

[92] S. Wu, D. Jiang, B. Ooi, and K. Wu, "Efficient B-tree Based Indexing for Cloud Data Processing," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, 2010.

[93] D. Abadi, "Data management in the cloud: Limitations and opportunities," *Data Engineering*, p. 3, 2009.