5 Cache Cluster High Priority based Framework

5.1 Introduction

In this section I want to introduce a High-Priority based cache clustering algorithm, in previous section I tried to describe the platform that is used for this algorithm we address the problem of buffer management in a OODBMS when the workload consists of transactions of different High-Priority levels. We present Priority-Cards. A new buffer management algorithm that uses hints provided by the OODBMS access methods. The performance of Priority-Cards is compared to that of priority buffer management schemes introduced earlier for a variety of workloads. Our simulation results indicate that Priority-Cards performs consistently better than simple LRU-based algorithms. Furthermore, our algorithm approaches (and in some cases surpasses) the performance of highly sophisticated algorithms that require much more information to be provided to the buffer manager.

5.2 Related Work

Priority scheduling has recently become an area of increased interest to the database community [SIGM88, Abbo88, Abbo89, Care89, and Hari90]. Applications that require different levels of system response for different transactions (for example, a system that is designed to provide faster service to interactive jobs than to batch jobs) can benefit from priority scheduling at the DBMS resources, as shown in [Care89]. Several data-intensive applications such as computer-aided manufacturing, stock trading, and command and control systems may require real-time response, which can also be supported with the help of priority scheduling at the resources of the DBMS [SIGM88, Abbo89].
5.3 Problem issues

The use of priority in DBMS resource scheduling may lead to an increase in the extent to which buffer management impacts system performance compared to its impact in conventional database Systems. Unpredictable bursty arrivals of high-priority transactions may force a priority-oriented Regions where the total load on the buffer pool (i.e., the sum of the buffering requirements of transactions of all priority levels) exceeds the buffer pool capacity. In these operating regions, priority-based load control and buffer allocation policies will be required, as the use of conventional load control and allocation techniques may lead to situations of priority-inversion,” where high-priority transactions are forced to wait while low-priority transactions are allowed to make progress. Furthermore, the set of concurrently active transactions in these operating regions may include transactions of different priority levels. In this scenario, then, priority-based buffer replacement policies may also be required in order to provide preferential service to high-priority transactions.

We anticipate that all aspects of buffer management (load control, allocation, and replacement) will become both more complex and more significant when priority is used in scheduling DBMS resources. ‘Several interesting new issues arise when buffer management decisions have to include priority considerations. One such issue is the tradeoff between the overheads introduced as a consequence of the use of priority and the advantages provided to high-priority transactions. If a buffer containing data accessed by a transaction is replaced as a consequence of priority, its data may have to be re-read from disk once the transaction resumes execution. The total load on the system may increase purely as a consequence of the use of priority in buffer management, and alternative priority-based buffer replacement and allocation policies may result in different relative increases in system load.

A second issue of interest is the extent to which information about the workload can be used by the buffer manager to improve system performance in the presence of High-Priority. Existing buffer management schemes assume different levels of information about transactions’ data access patterns [Effe84, Teng84, Chou85, and Sacc86]. In this section, we introduce a new buffer management algorithm that makes use of hints provided by the database access methods (as in the Starburst buffer manager [Haa&O] and DB2 [Teng84]). This new algorithm, called ‘Priority-Cards,” uses these hints to make priority-based buffer management decisions while trying to minimize the priority induced overhead on the system.
A third issue in priority-based buffer management is intertransaction buffering interference across priority levels. For example, update-intensive transactions may quickly make large numbers of buffers “dirty,” making them unavailable for replacement until they are written out to disk. The performance of high-priority transactions can thus be affected adversely by low priority updates. Another example of inter-transaction effects in the presence of priority may occur when high-priority sequential scans quickly replace a large number of buffers with pages that are accessed just once, unnecessarily depriving lower priority non-sequential transactions of buffers that need to be accessed repeatedly. Priority-based buffer management policies should be designed to minimize these effects. Finally, the importance of using priority-based buffer management in a DBMS that already uses priority at the CPUs and the disks may itself be open to question. In this research work, I investigated all these issues deeply and tried to develop an algorithm for them.

5.4 Algorithm assumptions and work steps

The design of the Priority-Cards algorithm was motivated by the intuition that the use of simple page-level information by the buffer manager (as in [Teng84, Haas90]) may improve system performance over that provided by simple LRU-based approaches. Such a performance improvement acquires great importance in priority-based systems, where significant I/O overheads may result as a consequence of using priority in buffer management decisions. Improvements in performance (relative to simple LRU-based approaches) may also be obtained by using sophisticated DBMIN-like algorithms, as shown in [Care89], but only at the cost of added system complexity; our goal is to exam in whether such complexity is really required. In order to take a detailed look at the issues involved in priority-based buffer management in the presence of mixed workloads, I have implemented a simulation model of an OODBMS that uses priority scheduling. In this work, we use this simulation model to compare the performance of our new algorithm with Priority-LRU and PriorityDBMIN. My primary objective in this analysis is to explore the extent to which our algorithm can surpass the performance of Priority-LRU, and how close we can get to the performance of Priority-DBMIN. We also conduct experiments that shed light on the other priority-related buffer management issues raised earlier.

5.5 HIGH -PRIORITY -BASED BUFFER MANAGEMENT ALGORITHMS

In this section I present the Priority-Cards algorithm. Our assumptions about buffer management are outlined first. We then describe Priority-Cards and briefly review Priority-
LRU and Priority-DBMIN, the two algorithms presented in [Care89]. Our scheme for handling dirty data, which is common to the three algorithms, is described next; and we conclude the section with a summary of the key differences between the three algorithms.

5.5.1 Buffer Management Assumptions

A page is assumed to be fixed (or pinned) in the buffer pool during the interval when a transaction is processing the data on the page. As soon as the transaction has finished processing the page, it unfixes it. Fixed pages cannot be chosen as buffer replacement victims. The owner of a resident page is the transaction with the highest priority among the executing transactions that have accessed the page since it was brought into memory. The buffer manager associates a timestamp with each resident page in order to keep track of the regency of usage of pages.

Each time the data in a buffer is accessed or updated, a global counter is incremented and its new value is inserted as the timestamp of the page. Thus, the larger the value of the timestamp of a page, the more recently the page was accessed. Pages in the free list (and the dirty list, described at the end of this section) are kept in LRU order using their timestamps. Based on the number of buffers available, a transaction may be admitted to the system right away, or it may be blocked initially. Transactions blocked outside the system are queued in order of priority. Once a transaction is allowed to begin execution, it continues until it commits, is aborted as a result of concurrency control, or is suspended.’ A transaction is said to be suspended by the buffer manager if it is temporarily prohibited from making further buffer requests; the buffers owned by the transaction are freed.

The buffer manager considers reactivating suspended transactions at the same decision point that it considers admitting blocked transactions, which is whenever a running transaction completes or aborts. A reactivated transaction resumes its execution at the point where it was suspended. A transaction checks whether it has been chosen as a suspension victim at instants when it has no pages fixed. We call such instants “suspension-safe” points. At suspension-safe points, the transaction “volunteers” to let all of its buffers be stolen by transactions of higher priority. Such instants occur normally during the execution of a transaction. In a sequential scan, for example, they occur at the point when the transaction unfixes one page and is about to request that another be fixed. In addition, a priority DBMS should be designed so that transactions periodically “come up for air” and check if they need to give up their buffers.
5.5.2 The Priority-Cards Algorithm

As its name suggests, Priority-Cards makes use of hints (provided by the DBMS access methods) that indicate whether a particular data page should be retained in memory in preference to other data pages. The basic ideas underlying Priority-Cards are the following:

- As discussed in [Teng84, Haas90], it is possible to classify the pages referenced by a transaction into two groups: pages that are likely to be re-referenced by the same uansaction (such as the pages of the inner file in a nested-loops join), and pages that are likely to be referenced just once (such as the pages of a file being scanned sequentially). The pages that are likely to be referenced are called favored pages, and the others are called normal pages. We assume that whenever a request for a page is made to the buffer manager, the buffer manager is informed whether the requested page is favored or normal.

- The favored pages of a transaction should be kept in the buffer pool as long as the transaction needs to reaccess them; each normal page should be made available for replacement as soon as the transaction unites it. When searching for replacement victims, normal pages should therefore be considered before favored pages.

- If it becomes necessary to choose a favored page as a replacement victim, the most-recently-used (MRU) policy should be used to choose the victim. As discussed in [Chou85], MRU is a better approach than LRU when choosing replacement victims from a set of pages that are being repeatedly. Looped over, and favored pages are likely to fall into this category.

The Priority-Cards algorithm combines these ideas with the notion of priority as follows.

**Buffer Pool Organization:** Buffers are organized into “transaction sets,” where a transaction set consists of all of the buffers owned by a single transaction. Transaction sets are arranged in priority order, with regency of arrival of the owner transaction being used to break ties if there are multiple transactions of the same priority. In the buffer pool configuration shown in Figure 5.1, there are three transactions (T1, T3, and T2), three priority levels, and no free buffers. A transaction set consists of two kinds of buffers: the buffers currently fixed by the owner (marked by the letter “F” in Figure 5.0). And buffers containing unlied favored pages of the owner (marked by the letter “U” in Figure 5.0). The unfixed favored pages are maintained in MRU order with the help of buffer timestamps. Note that a transaction set contains no unfixed normal pages; whenever a normal page is unfixed, it is freed.
**Transaction Admission:** Transactions are required to estimate the maximum number of pages that they will need to fix concurrently, and the buffer manager keeps track of the sum of these “fixing requirements” for all active transactions. If admitting a newly arrived transaction does not cause this sum to exceed the size of the buffer pool, the transaction is admitted. Otherwise, if there are running transactions of lower priority than that of the new arrival, the one(s) with the lowest priority among them are suspended until there are enough buffers for the new arrival, or until no lower priority transactions remain4 If no remaining transactions are of a priority less than the new arrival, then the new arrival is forced to wait outside the DBMS.
Figure 5.1: Example of Priority-Cards Buffer Pool Organization.

5.5.3 Buffer Replacement and Allocation:

When a buffer miss occurs and there is no free page available, the buffer manager first attempts to get a replacement victim from among the unfixed favored pages of transactions of lower
priority than the requesting transaction. The buffer pool searches its transaction sets in inverse priority order, starting from the lowest priority transaction, looking for unfixed favored pages. It stops searching on either of the following conditions:

(1) It finds a transaction of lower Priority than the requesting transaction with an unfixed favored page; or

(2) It has reached a transaction of a priority equal to or greater than that of the requesting transaction.

In case (1), it chooses the most recently unfixed favored page of the lower-priority transaction as the replacement victim. In case (2). It chooses the most recently unfixed favored page (if any) of the requesting tranaction itself. Note that this means that transactions cannot steal buffers from other transactions of the same priority; thus the replacement policy for favored pages is focal rather than global. If no replacement victim is available, then the outstanding request is queued in the Buffer Waiting Queue. Furthermore, if there are running transactions of lower priority, the transaction with the lowest priority among them is suspended. Continuing the example of Figure 2.1, if T1 makes a buffer request for page P6, which is not in the buffer pool, the buffer manager will start its search for replacement at T2. Finding no unfixed buffer in T2’s transaction set it will look at T3’s transaction set and find P63 as the replacement victim. Had there been no unfixed pages of priority 1 or 2, then P100 would have been chosen as the replacement victim. To summarize, Priority-Cards has a focal MRU replacement policy for favored pages, and a global LRU replacement policy for normal pages. (Recall that normal pages are placed on the free list at unfix time, and that the free list is maintained in LRU order.)

### 5.5.4 The Priority-LRU Algorithm

In Priority-LRU, the prioritized version of Global LRU, the buffer pool is organized dynamically into priority levels as described in [Care89]. Each priority level consists of pages whose owners have the same priority, and the pages within a level are arranged in LRU order. The transaction admission policy for Priority-LRU is the same as that for Priority-Cards. The key idea of the Priority-LRU replacement policy is that the least recently unfixed page of the lowest priority should be chosen as the victim. If there are no free buffers, the search for a replacement victim starts at the lowest priority queue, where we check whether unfixed candidates are available. If such candidates are found, the least-recently-used candidate is chosen as the victim. If no candidate is found at this priority level, we move up one level, and
we repeat the process until we have either found a victim, reached a priority level that exceeds
that of the requesting transaction, or exhausted the search. If no victim is found, and there are
transactions of lower-priority running, the lowest priority transaction is suspended as in
Priority-Cards.

5.5.5 The Priority-DBMIN Algorithm

As discussed in [Chou85], the primitive operations (e.g. Selections, joins) of transactions in a
relational DBMS can be described as a composition of a set of regular reference patterns such
as sequential scans and hierarchical index lookups. These patterns are known to the query
optimizer. The DBMIN buffer management policy makes use of this information in the
following way:

A set of buffers (called a “locality set”) is allocated to each transaction for each file accessed
by it. The optimum size of each locality set and the optimum replacement phony to be ‘used
within a locality set are supplied to the buffer manager by the optimizer. DBMIN guarantees
that each transaction that is allowed to enter the system has the optimum number of buffers
available to it. And the optimum replacement policy is used within each locality set.

Priority-DBMIN, the prioritized version of DBMIN, also allocates buffers to transactions in
locality sets. A transaction is allowed to enter the system only if its optimal-sized locality sets
can be accommodated in the buffer pool. Otherwise, if there are transactions of lower priority
than that of the arriving transaction in the system, they are suspended in reverse priority order
until sufficient buffers become available for the new arrival. As in the original DBMIN
algorithm, Priority-BMIN uses the optimize Supplied optimum replacement policy within each
locality set.

5.5.6 Dirty Data

In all three algorithms, a process called the asynchronous write engine [Teng84] is responsible
for Rushing dirty buffers to disk. When a transaction frees a buffer, the buffer is inserted into
the free list if it is clean (i.e., if it has no update that has not been written to disk). If the data in
the buffer has been updated, the buffer is placed in a queue called the dirty list. The write engine
is activated periodically; it also wakes up whenever a buffer miss OCCUTS and the free list is
empty. The engine flushes each page in the dirty list that is sufficiently “old” in terms of its
regency of use. ‘Requests to write dirty buffers are asynchronous. This may result in some
buffer requests having to wait until a buffer is flushed to disk. Write requests to the disk are therefore assigned a priority equal to the highest possible transaction priority. When its I/O is completed, a dirty buffer is marked clean and placed on the free list, and if there are any buffer requests pending, the highest-priority request is serviced. When choosing replacement victims, dirty data is avoided as long as possible. That is, if a buffer that would normally be a candidate for replacement is dirty, we ignore it in our search for replacement victims unless all candidate buffers are dirty.

5.5.7 Discussion

In summary, the key features of Priority-Cards that distinguish it from the other algorithms discussed are the following:

- By realizing which pages are normal, Priority-Cards is able to free more buffers earlier in the course of a transaction’s execution than Priority-LRU. In this respect, Priority-Cards behaves similarly to Priority-DBMIN.
- When choosing replacement victims from among non-free pages, Priority-LRU chooses the least-recently-unfixed page; Priority-Cards chooses the most-recently-unfixed page. MRU is likely to be a better policy when the replacement victim is part of a set of pages that are being looped over; in cases where pages are reaccessed randomly, the performance differences between MRU and any other replacement policy are negligible [Chou85].
- Priority-Cards’ replacement policy ensures that the favored pages of a transaction can be stolen only by a transaction of higher priority: in Priority-LRU, transactions of the same priority

5.6 MODELING A PRIORITY-ORIENTED OODBMS

In this section, we describe our performance model of a priority-oriented OODBMS. The model, which we implemented using the DeNet simulation language, consists of five components: the database itself; a Source, which generates the workload of the system; a Transaction Manager, which models the execution behavior of transactions; a resource Manager, which models the CPU, I/O, and buffer resources of the system; and a Concurrency Control Manager, which implements the details of a particular
concurrency control algorithm. Since we will be using workloads where concurrency control is not an
issue, we will not discuss the Concurrency Control Manager further. (As described in
Section 4, our workloads consist either of read-only transactions with data sharing, or
updates without data sharing.) In most respects, our model is similar to the model
described previous. Therefore, we describe its components very briefly here; see
[Care891 for more details.

5.6.1 Modeling the Database

The database is modeled as a collection of relations. Each relation in turn is modeled as a
collection of pages. Indices (clustered or unclustered B+ Trees) on the base relations are ‘It is
advisable to allow Priority-LRU to do this, as many of the pages owned by a transaction are
likely to be accessed just once. Included in the database model. The parameters for the database
model are summarized in Table 5.1.

5.6.2 The Source Module

The Source module is the component responsible for modeling the workload of the
dBMS. Table 5.2 summarizes the key parameters of the workload model. A transaction
may belong to any one of NumClasses classes, and it may have any one of
NumPriorities priority levels. The model is that of an open system, and transactions of
each <classi, rijyj> combination arrive at the system in a Poisson process with a mean
arrival rate of ArrRateij. Transactions can be single-relation selects, single relation
select-updates, or two-relation select-joins; the type of a transaction of class i and
priority j is controlled by TrmTypeij. Selections can be performed via sequential scans
or index scans, and we model three join methods: nested-loops joins, classic hash joins,
and index joins. For each transaction type (selection, join, or update) of a particular
priority level, an execution plan is provided in the form of a set of parameters. For
selections, the access path and the mean selectivity are provided as parameters. The
actual selectivity is varied uniformly over the range [Selectivityijk/2, 3*Selectivityijk/2]. For select-joins, the join method and the inner and outer relations
are provided in addition to the selection parameters. For select-updates, the probability
of updating a page is specified as the parameter pdateProbijk. Finally, times spent at
the CPU for processing or updating a page are uniformly distributed: the CPU time per data page of relation k varies uniformly over the range \([\text{DataPageCPU,} jk12,]\).

<table>
<thead>
<tr>
<th>NumRelations</th>
<th>Number Of Relations in database</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelSize(_i)</td>
<td>Number Of Pages In relations i</td>
</tr>
<tr>
<td>Indexed(_i)</td>
<td>Whether relation i has an index</td>
</tr>
<tr>
<td>Indextype(_i)</td>
<td>Type of index (clustered/nonclustered)</td>
</tr>
<tr>
<td>Fanout(_i)</td>
<td>Fanout of internal nodes of index</td>
</tr>
</tbody>
</table>

**Table 5-1: Database Model Parameters.**

<table>
<thead>
<tr>
<th>Overall arrival parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NumClasses</strong></td>
</tr>
<tr>
<td>Number Of Transaction classes</td>
</tr>
<tr>
<td><strong>NumPriorities</strong></td>
</tr>
<tr>
<td>Number Of Transaction Priority levels</td>
</tr>
<tr>
<td>ArrRates&lt;sub&gt;ij&lt;/sub&gt;</td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>TransType&lt;sub&gt;ij&lt;/sub&gt;</td>
</tr>
<tr>
<td>Outer&lt;sub&gt;ij&lt;/sub&gt;</td>
</tr>
<tr>
<td>Inner&lt;sub&gt;ij&lt;/sub&gt;</td>
</tr>
<tr>
<td>AccessPath&lt;sub&gt;ijk&lt;/sub&gt;</td>
</tr>
<tr>
<td>Selectivity&lt;sub&gt;ijk&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Table 5-2: Workload Model Parameters.

5.6.3 The Transaction Manager Module

The Transaction Manager is responsible for accepting transactions from the Source and modeling their execution. For each page accessed by the transaction, the Transaction manager sends a read (or write) request to the Resource Manager; the Resource Manager informs the Transaction Manager when the request is completed. The Resource Manager also informs the Transaction Manager when a transaction is suspended or reactivated. When the Resource Manager decides to reactivate a suspended transaction, the transaction Manager ensures that the reactivated transaction resumes execution at the point where it was suspended.

5.6.4 The Resource Manager Module

The Resource Manager controls the physical resources of the DBMS, including the CPU, the disk, and the buffer pool in main memory. Three versions of the Resource Manager have been implemented, supporting the Priority-LRU, Priority-DBMIN. And Priority-Cards algorithms, respectively. Resource Manager Parameters are summarized in Table 5.2.

5.6.5 CPU and Disk Models:

The DBMS has MumCPUs CPUs and a single priority queue for outstanding CPU requests. The actual CPU where a request is processed is selected at random from among the idle CPUs, if any. The length of each CPU request from a transaction is its per-page CPU processing time; each transaction voluntarily gives up the CPU after
processing or updating one page, as in the priority-based round robin CPU scheduling scheme described in [Care89]. There are \( \text{NumDisks} \) disks in the system, with requests at each disk being priority-scheduled according to the prioritized elevator algorithm [Care89]. We model the data as being uniformly distributed across all disks and across all tracks within a disk. The total time required to complete a disk access is computed as the sum of its seek time, rotational latency, and transfer time components. As in [Bitt88,Care89], there is a square root relationship relating seek time to seek distance, and the rotational latency and transfer time are together modeled as a single parameter called DiskConFt.

### 5.6.6 Buffer Manager Models:

The Buffer Manager component of the resource manager encapsulates the details of the buffer management scheme employed. The number of page frames in the buffer pool is specified as NumBuffers. A separate buffer manager component has been implemented for each buffer management algorithm studied.

<table>
<thead>
<tr>
<th>Table 5-3: Parameters of the Resource Manager.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumCPUs</td>
</tr>
<tr>
<td>NumDisks</td>
</tr>
<tr>
<td>NumTracks</td>
</tr>
<tr>
<td>DiskConst</td>
</tr>
<tr>
<td>SeekFactors</td>
</tr>
<tr>
<td>Numbuffers</td>
</tr>
</tbody>
</table>
5.7 EXPERIMENTS AND RESULTS

In this section, we present performance results for the priority buffer management algorithms described earlier. In [Care89] it was shown that results for two priority levels can be generalized to multiple priority levels, so we consider just two priority levels: “low” versus “high” priority. In addition, our workload consists of three types of single-query transactions: “looping”, “random reaccess (RR)”, and “scanting” transactions. As their names suggest, looping transactions (such as nested-loops joins) reaccess some of their pages sequentially a number of times, RR transactions (such as hash joins) randomly reaccess some of their pages, and scanning transactions (such as clustered-index selections) touch each page just once. Looping transactions and scanning transactions represent two ends of the spectrum of buffer access characteristics typical in relational databases, while RR transactions represent the middle.

5.7.1 Performance Metrics

As discussed earlier, we use an open queuing system to model the DBMS. Our primary performance metric will be the average response time ratio (RTRatio) for transactions at each priority level. We define the RTRatio of a transaction as the ratio of the actual response time of the transaction to its estimated response time on an unloaded system with an infinitely large buffer pool. A transaction’s response time is computed by subtracting the time at which the transaction commits from the time at which it was submitted to the DBMS. The response time of the transaction in an unloaded system is estimated by summing the CPU requirements associated with the page accesses of the transaction and by assuming one I/O per distinct page referenced by the transaction. That is, only one I/O is assumed for a page, whether it is touched just once by a transaction or accessed repeatedly. The RTRatio of a transaction, then, reflects the effects of the finite size of the buffer pool and the presence of competing transactions on the response time of the transaction. As the load on the system increases, contention for buffers causes increased I/O (and an increase in the time spent waiting outside the system) while contention for disks and CPUs causes increased disk and CPU waiting times. These factors tend to increase the RTRatio of transactions.

On the other hand, if there is significant data sharing, the RTRatio of a transaction would tend to be reduced because part of the transaction’s read and write sets would already be in main prioritized elevator disk scheduler has two priority queues, one for each priority level. The factor relating seek distance to seek time is 0.6 milliseconds, so the expected disk access time
is between 15 and 30 milliseconds. As stated in Section 1, the operating region of greatest interest to us is when the combined buffer requirements of all transactions exceeds the capacity of the buffer pool. In order to simulate the behavior of the system in this region of operation without incurring excessive simulation costs, we kept the buffer pool relatively small in our experiments. Thus, there are 50 buffer frames in the buffer pool in the base experiment. One point that should be noted here is that, from a performance perspective, it is not the actual size of the buffer pool that is most significant. Instead, two ratios are more important: the ratio of the combined buffering requirements of concurrent transactions to the size of the buffer pool\'O, and the ratio of the size of the buffer pool to the size of the database. For this study, we vary the first of these ratios by varying the arrival rate of high-priority transactions in all our experiments. We study the effects of varying the second ratio in Experiment 2 by changing the database size.

5.7.2 Workload Parameter Settings:

The workload for the base experiment consists of two types of eactions. Looping transactions consist of select-joins, with the result of a selection using a clustered index on a 500-page outer relation being joined to a smaller inner relation. The selectivity of the outer relation selection varies uniformly between 0.5% and 1.5%. The inner relation is chosen uniformly from among the 30 small relations of sizes between 3 and 5 pages. Page Accesses is the expected number of page accesses for the transaction, with repeated references being counted as one access each time.” Scanning transactions are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Looping</th>
<th>Scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransType</td>
<td>Select-join</td>
<td>Scan</td>
</tr>
<tr>
<td>JoinMethod</td>
<td>Nested Loops</td>
<td>-</td>
</tr>
<tr>
<td>RelSize&lt;sub&gt;1&lt;/sub&gt; (outer)</td>
<td>500-page</td>
<td>1000-page</td>
</tr>
<tr>
<td>RelSize&lt;sub&gt;2&lt;/sub&gt; (inner)</td>
<td>3-5 pages</td>
<td>-</td>
</tr>
<tr>
<td>AccessPath&lt;sub&gt;1&lt;/sub&gt;</td>
<td>Cl. Idx. Scan</td>
<td>Cl. Idx. Scan</td>
</tr>
<tr>
<td>AccessPath&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Seq. Scan</td>
<td>-</td>
</tr>
<tr>
<td>Selectivity&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>IndexPageCPU</td>
<td>4 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td>DataPageCPU</td>
<td>4 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td>Page Accesses</td>
<td>43</td>
<td>13</td>
</tr>
<tr>
<td>Locality Set Sizes (index, outer, inner)</td>
<td>1, 1, 3-5</td>
<td>1, 1</td>
</tr>
<tr>
<td>Repl. Policies</td>
<td>MRU, MRU, MRU</td>
<td>MRU, MRU, -</td>
</tr>
<tr>
<td>Filing Requirements</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 5-4: Workload Parameter Settings

Than with Priority-DBMIN. Finally, Global-Hints provides better performance than Priority-LRU, but is significantly worse than Priority-Cards. Priority-DBMIN provides better performance than Priority-Cards because the admission control policy of Priority-DBMIN uses its knowledge of the optimum number of buffers required for each transaction, while Priority-Cards’ admission control policy cannot distinguish between the buffer requirements of looping transactions and those of scanning transactions. Consequently, Priority-Cards allows looping transactions to enter the system even when their loops cannot be guaranteed to fit in the buffer pool. This results in a higher buffer miss ratio for Priority-Cards than for Priority-DBMIN, and causes the system to become unstable at a lower arrival rate. When we move to the Global-Hints algorithm from Priority-Cards, the local search for MRU replacement victims is replaced by a global search. This change results in a significant performance degradation for the following reason: In Priority-Cards, for looping transactions, once a transaction is able to obtain enough buffers to keep its loop (the inner relation in the nested-loops join) in memory, it proceeds quickly since it never has to give up any buffers. (Recall that no transaction can steal buffers from any other transaction in Priority-Cards for this workload.) In contrast, Global-Hints steals buffers indiscriminately from all transactions, often depriving looping transactions that have their entire working set in memory of some of their favored buffers.

This causes a significant increase in disk activity for Global-Hints as compared to Priority-Cards. As a result, the system becomes unstable for Global-Hints at a foreground arrival rate of approximately 5 transactions/second, while Priority-Cards keeps the system stable until an arrival rate of about 7.5 transactions/second. Finally, the difference between the curves for Global-Hints and Priority-LRU is caused by two features of Global-Hints. Firstly, MRU is a better search strategy for buffer replacement than LRU when the workload contains looping transactions.

Secondly, Global-Hints frees normal pages as soon as it unfixes them; Priority-LRU does not. This results in favored pages being chosen as replacement victims more frequently in Priority-LRU than in Global-Hints. This experiment shows that even in the absence of priority, Priority-
Cards’ local MRU replacement strategy for favored pages provides significantly better performance than Priority-LRU for our base workload, and matches the performance of Priority-DBMIN over a wide range of arrival rates. It also isolates the relative impact of the following buffer management features: admission control, which causes the difference between Priority-DBMIN and Priority-Cards; local vs. global search for replacement victims, which causes the difference between Priority-Cards and Global-Hints; and the use of MRU vs. LRU search strategies in a looping workload, which is the major factor causing the gap between the curves for Global-MRU and Priority-LRU. We can now begin our investigate the performance of the system when the workload consists of transactions of different priority levels.

5.7.3 Experiment 1: The Base Experiment

In this experiment, we study the impact of using priority in buffer management for our base workload. Figure 5.2 shows the RTRatios for high-priority transactions for five buffer management algorithms: Priority-DBMIN. Priority-Cards, Priority-LRU.Global-Hints, and Global-LRU. RTRatios for low-priority transactions are shown in Figure 5.3. We explain the results of the base experiment in detail in order to provide insights into the important issues involved; this will allow us to present the results of subsequent experiments more briefly. Comparing the curves for each algorithm in Figure 5.1 with the corresponding curves in Figure 5.2, we see that we have achieved the primary goal of priority scheduling, which is to provide a higher level of performance for high-priority transactions.

For example, the system remains stable for a foreground arrival rate of up to 12 transactions/second in Figure 5.2 for Priority-Cards, while the system saturates at a foreground arrival rate of about 6 transactions/second in Figure 5.1 for the same algorithm. Of course, there is a corresponding price which is paid by low priority transactions, as is made clear by comparing Figures 5.1 and 5.3. A secondary goal of priority scheduling is to minimize the penalty imposed on low-priority transactions; distinctions between the different algorithms in this respect will become clear as we describe subsequent experiments. From Figure 5.2, we see that the behavior of Priority-Cards for high-priority transactions is very close to that of Priority-DBMIN, and both are superior to the other three algorithms.

An interesting feature of the behavior of these two algorithms is the tradeoff between the time spent by transactions waiting outside the system in Priority-DBMIN and the time spent inside the system competing for resources in Priority-Cards. Priority-Dublin’s conservative admission
policy causes the transactions ‘mean time spent waiting outside the system to increase more as the load on the system is increased than does Priority-Cards ‘liberal admission policy. However, since there are more transactions within the system in Priority-Cards than in Priority-DBMIN, the buffer m77iss ratios and the mean waiting times at the disks are higher for Priority-Cards than for Priority-DBMIN. This tradeoff will be referred to again in the following sections, where we will refer to it as the “conservative-liberal (C-L)” tradeoff. In Figure 5.4, we present the mean number of transactions (both total and high-priority) that are allowed to run concurrently by the two algorithms. Figure 5.5 shows the mean normalized Disk-Time and the mean normalized OutWaitTime for high-priority transactions for the two algorithms. DiskTime is the time spent by a transaction at the disks, including the actual I/O service time and time spent waiting for disk service; OutWaitTime is the time spent by a transaction waiting outside the system. DiskTime and OutWaitTime for a transaction are normalized by dividing each of them by the expected response time of the transaction in an unloaded system.

Figure 5.4 indicates that Priority-Cards allows up to 18 high priority transactions into the system, while Priority-DBMIN limits the number of concurrent high-priority transactions to 12. Figure 5.5 shows the consequences of this. The DiskTime curves reflect the relative disk contention in the two algorithms, and Priority-DBMIN is the clear winner in limiting disk contention; both the average buffer miss ratio and the average disk waiting times per transaction are higher in Priority-Cards than in Priority-DBMIN. In contrast, Priority-Cards is the winner in limiting OutWaitTime up to an arrival rate of up to 11 transactions/second. As the load is increased beyond this, however, the disk utilization nears 100% for Priority-Cards, causing the system to saturate and Priority-Cards’ OutWaitTime to exceed that of Priority-DBMIN. Priority-DBMIN’s conservative admission control policy enables it to keep the system stable for arrival rates of up to 13 transactions/second. Figure 5.4 also reveals an important point about the range of operation of greatest interest to us.

There is a gap between the total number of concurrent transactions and the number of high priority transactions for most arrival rates shown; this gap corresponds to the number of low-priority transactions running in the system. When the curves for the total number of transactions in the system flatten out, the buffer pool has become fully utilized, but Figure 5.4 shows that there are still significant numbers of low-priority transactions running in the system. It should be clear that priority-based buffer replacement policies will be most useful in this range of operation, since buffers owned by low priority transactions can be stolen by high-priority transactions. In Figure 5.4 we also see that as the arrival rate of high-priority transactions
increases, low-priority transactions are gradually displaced by high-priority transactions (due to the use of priority based Admission control policies) until finally only high-priority transactions remain active. Thus, both priority-based admission control and priority-based buffer replacement have an important role in determining performance over a fairly wide range of arrival rates.

Figure 5.2 shows us that at low loads, all the algorithms provide similar levels of performance to high-priority transactions. As the high-priority load increases, the curves for Global-LRU, Global-Hints, and Priority-LRU soon branch away from the curves for Priority-Cards and Priority-DBMIN. Global-LRU performs worst of all: it does not distinguish between transactions of high and low priorities, and it uses the LRU criterion for replacement. As for Global-Hints, its ability to distinguish between favored and normal pages (and its use of MRU) actually allows it to perform better than Priority-LRU at very low high priority loads. As the load is increased, however, Priority-LRU’s protection of high-priority buffers begins to have a greater impact, since more of the buffers are now owned by high-priority transactions. The RTRatio of high-priority transactions remains lower for Priority-LRU than for Global-Hints until the system becomes unstable for both algorithms at a high-priority arrival rate of approximately 8 transactions/second. In Figure 5.3, the curves for Priority-Cards and Priority-DBMIN are fairly close. This is because the criteria used for suspending low-priority transactions differ in these two algorithms, making the C-L tradeoff more even for low-priority transactions than it is for high-priority transactions. (High priority transactions cannot be suspended in either algorithm.) Priority-Dublin suspends a low-priority transaction immediately when one of its unfixed buffers is required by a high-priority transaction. In contrast, Priority-Cards does not suspend low priority transactions as frequently as Priority-DBMIN does. It allows them to continue execution as long as their “fixing requirements” can be satisfied; of course, this increases the low priority load on the disks. As long as there is sufficient disk capacity to handle this increased low-priority load in Priority-Cards, the two algorithms provide similar performance for low priority transactions.

Figure 5.3 also shows that there is very little difference in the performance provided by the LRU algorithms (Global-LRU and Priority-LRU) for low-priority transactions. Priority-LRU steals buffers from low-priority transactions in preference to depriving high-priority transactions of their buffers, so one might expect Priority-LRU to provide worse performance for low-priority transactions. Recall, however, that the CPUs and the disks use priority
scheduling in these experiments; also, in the range of arrival rates for which the system is stable for low-priority transactions, there are relatively few high-priority transactions in the system.

The globally least-recently-used buffer is therefore quite likely to belong to a low-priority transaction rather than to a high-priority transaction. This is why the curves for Priority-LRU and Global-LRU are so close to each other in Figure 5.3. Low-priority transactions perform better under the Global-Hints algorithm than under the two LRU algorithms in Figure 5.3 because their buffer miss ratios are lower due to the use of Murtha reason that Global-Hints performs worse than Priority-Cards there, even for low-priority transactions, is again related to the use of priority at the CPUs and at the disks. As the arrival rate of high-priority transactions increases, Global-Hints hurts high priority transactions more than Priority-Cards does (since Global-Hints ignores priority in choosing replacement victims). Consequently, more and more of the system’s disk capacity is used to satisfy high-priority transactions in Global-Hints. This makes the disk waiting times of low-priority transactions higher, causing the system to become unstable for low-priority transactions at a lower load in Global-Hints than in Priority-Cards. The base experiment confirms the result of [Care891 for our mixed workload: the use of priority in buffer management is a clear win (independent of the algorithm) if the response time of high-priority transactions is the main criterion of system performance.

However, the conclusions are more mixed for low priority transactions: their performance may be worse for some priority-based buffer management algorithms (e.g., Priority-LRU) than for algorithms that do not consider priority in replacement decisions (Global-Hints). The base experiment also shows that for both low-priority and high-priority transactions, the performance provided by Priority-Cards is significantly better than the performance provided by Priority-LRU, and that Priority-Cards performs almost as well as Priority-DBMIN for both priority levels. In subsequent experiments, we limit ourselves to showing the relative behavior of the three priority-based algorithms.
Figure 5.1: NO PRIORITY

Figure 5.2: high Priority (Base Experiment)
Figure 5.3: low priority (Base Experiment)

Figure 5.4: Number of Concurrent Transactions
Figure 5.5: High Priority (Medium Data Sharing)
Figure 5.6: Disk time and Out Wait Time

Figure 5.7: Low Priority (High Data Sharing)
Figure 5.8: High Priority (High Data Sharing)

Figure 5.9: High Priority (100% scanning process)
5.7.4  Experiment 2: Varying Relative Buffer Pool Size

In this experiment, we reduce the size of the database while keeping all other parameters the same as in the base experiment. Thus. The ratio of the size of the buffer pool to the size of the database is higher in this experiment than in the base experiment. Increasing the relative size of the buffer pool in this manner results in increased data sharing: i.e., data brought into the bufferpool at the request of one transaction is more likely to be found in memory when it is accessed by other transactions. When data accesses are distributed uniformly over the database, the extent of data sharing is inversely proportional to the database size if all other parameters are kept fixed. In the base experiment, the database consisted of 50 relations and a total of 15,120 data pages, while the buffer pool had 50 buffers (see Table 5.1).

This represents a fairly low level of data sharing (and we did not even take index pages into account when computing the sum of 15,120 pages!). We consider two levels of data-sharing in this experiment, where there are 25 relations in the database (five relations each of 1000 pages, 500 pages, 5 pages, 4 pages, and 3 pages), and one where there are just 5 relations (one of each size). When there are 25 relations in the database, the level of data sharing is double the level of data sharing in the base experiment; when there are just 5 relations, the level of data sharing is ten times that in the base experiment. In Figure 5.6, we present the Ratios of high-priority transactions for the three algorithms with 25 relations in the database. Figure 5.7 shows the RTRatios of low-priority transactions for the same database. As one would expect, the performance of all three algorithms improves over their performance in the base experiment. The key difference between the trends shown in Figures 5.4 and 5.2 is the fact that Priority-Cards now provides better performance than Priority-DBMIN at high loads.

As the level of data sharing increases, Priority-DBMIN’s conservative admission control policy proves to be more and more harmful: it simply underutilizes resources by failing to consider the possibility of data-sharing. The same conclusion holds true for low-priority transactions, where Priority-Cards consistently performs as well as or better than Priority-DBMIN. Figures 5.8 presents the RTRatios for high-priority transactions in the case where there are just 5 relations in the database. Priority-Cards and Priority-DBMIN provide the same level of performance for high-priority transactions until Priority-DBMIN’s admission control policy causes it to block transactions unnecessarily outside the system; beyond this point, Priority-Cards is better. As for Priority-LRU, one might initially expect that all three inner relations of the nested-loops joins (a total of 12 pages) would always remain in memory. And that it should therefore
perform as well as Priority-Cards. However, recall that there are now 1500 data pages (plus index pages) that are not part of inner relations. A large fraction of the page requests made to the buffer manager are for one of these 1500 pages. In Priority-Cards and Priority-DBMIN, the looping pages will remain in memory as long as there are some high-priority transactions that need them. Also, these two algorithms free normal pages as soon as they are unfixed, so it is quite likely that a free page will be available even when the load is high. In Priority-LRU, where transactions can steal buffers from other transactions of the same priority, and where looping pages are treated just like other pages, looping pages are frequently chosen as replacement victims. This is why the curve for Priority-LRU diverges from the other two in Figure 5.8. This experiment shows that as the relative size of the buffer pool increases, the performance of Priority-Cards improves to a greater extent than that of Priority-DBMIN. This is a consequence of the latter’s conservative admission control policy. Also, both Priority-Cards and Priority-DBMIN provide better performance than Priority-LRU, even under very high data sharing.

5.7.5 Experiment 3: Changing the Transaction Mix

In the base experiment, the high-priority workload consisted of a mix of an equal number of looping and scanning transactions. In this experiment, we vary the proportion of looping transactions in the high-priority workload while keeping all other parameters as in the base experiment. (The low-priority workload still consists of a 50% looping, 50% scanning mix.) Figure 5.9 shows the RTRatios for high-priority transactions when the high-priority workload consists entirely of scanning transactions. RTRatios for the low-priority transactions are shown in Figure 5.10. In Figure 5.9, the curves for the three algorithms coincide almost exactly. This is not unexpected, since the high-priority workload is insensitive to buffer replacement and the admission control criteria for the three algorithms coincide for scanning transactions. The interesting feature of this experiment is the relative performance of low-priority transactions, as shown in Figure 5.10: here, Priority-Cards perform better than Priority-LRU. The reason for this is that, unlike Priority-Cards, Priority-LRU does not free scanning pages as soon as they are unfixed. High-priority transactions. The interesting feature of this experiment is the relative performance of low-priority transactions, as shown in Figure 5.10: here, Priority-Cards perform better than
Priority-LRU. The reason for this is that, unlike Priority-Cards. Priority-LRU does not free scanning pages as soon as they are unfixed. High-priority transactions thus keep stealing looping pages from low-priority transactions in Priority-LRU. While the more appropriate action is to choose high-priority scanning pages as replacement victims. Figures 5.11 and 5.12 show the RTRatios for high-priority transactions and low-priority transactions, respectively, when the high-priority workload consists entirely of looping transactions. There is now a relatively larger difference between the performance of high-priority transactions for Priority-Cards and Priority-DBMIN than in the base experiment. The C-L tradeoff for high-priority transactions favors Priority-DBMIN in this experiment since a larger proportion of the high-priority workload now benefits from conservative admission control. However, note that the tradeoff is still quite even for transactions of low priority. This is because Priority-DBMIN now prevents more scanning low-priority transactions from entering the system than Priority-Cards does, and it also suspends them more often.

From Experiment 3. We learn that Priority-Cards tends to perform as well as or better than Priority-LRU, independent of the proportion of looping and scanning transactions. In particular, when the low-priority load is insensitive to buffer management, Priority-Cards provides much better performance than Priority-LRU for low-priority transactions. As the proportion of looping transactions in the high-priority workload increases, Priority DBMIN’s admission control policy makes it perform better than Priority-Cards for high-priority transactions, but the two algorithms provide very similar support for low-priority transactions.
Figure 5.10: HP Load (100 % scanning)

Figure 5.11: HP Load 100% Looping

Above and beyond from three experiments and results described above, I extensively experimented, researched and got research paper published in various International publications such as International Refereed Journal of Engineering and Science (IRJES),
International Journal of Advancements in Research & Technology and INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS(IJRAR) at various times during the study.

The topics researched, experimented and explored are here below

1. Object Oriented Database Management System for Decision Support System. Here I described how the OODBMS are beneficial for decision support systems in order to analyze data to make decisions for organizational benefits.

2. Column Based NoSQL Database, Scope and Future. In this paper I have described column based NOSQL databases, their features which can be beneficial to get the high performance systems for the good of human being.

3. To get best performance for analytics database systems by merging “in memory database” and “column oriented database” technologies. Paper describes the features of in memory databases and features of column oriented databases which leads to best performance. Paper also describes how these two technologies combined together to get best out of two. Microsoft and oracle has recently adopted the unique technologies but in their own way to get the best performances. Such as in memory OLTP concept makes its systems up to 45 times faster as per Microsoft claims. Similarly paper describes how no sql databases are 100 times faster, in this case mongo db.

4. Performance Management of In Memory Databases. Paper described the features of main memory databases which makes faster than that of traditional row oriented disk stored databases.

The above topics has been described in detail on following pages.

5.8 Object Oriented Database Management System for Decision Support System.

I researched and published research paper on the object oriented databases management system for decision support systems and the benefits of using the technology in order to the performance. Paper was published in June 2014 in International Refereed Journal of Engineering and Science ISSN (Online) 2319-183X, (Print) 2319-1821 Volume 3, Issue 6 (June 2014), PP.55-59.
To get best performance for an analytic system or data warehouse systems, two technologies, column oriented database management systems and main memory database management system can be combined to get advantages of these two. Both technologies give best performance to its opponent database system, for example Main memory database management systems are faster as they reside in main memory as compared to disk resident database systems. This is because main memory is faster in comparison to hard drive/disk. The performance of main memory database systems is 15-20% higher than that of disk resident database systems. Whereas column based database systems are faster than the row based database because in column based database systems, data is stored in columns and indexed as compared to row based database systems. Performance of column based database systems is 15-30% higher than that of row based database systems.

It is important to mention about static drive which way faster than conventional hard drives and provide similar performance as of main memory can improve and give best performance if used as hard drive to have database created and maintained on it.

By combining these two technologies, we can achieve 30-50% higher performance for analytics systems they needs to be high performing in analysis and computation. Whereas by having databases & data warehouse created and maintained on static drives will further improve performance to ~50-60%.

5.8.1 OBJECT ORIENTED DATABASES

Object oriented databases are the database in which information is stored and managed in the form of objects instead of data such as integer, characters, floats etc. Object oriented databases are also known as object databases. In object oriented databases, an object is first created, defined, named and then it is called at different point as per requirement similar to object oriented programming languages. Objects basically consist of the following:

- Attributes - Attributes are data which defines the characteristics of an object. This data may be simple such as integers, strings, and real numbers or it may be a reference to a complex object.
- Methods - Methods define the behavior of an object and are what was formally called procedures or functions.

Because of attributes and methods, object oriented databases contains both executable as well as data in the form of objects. An object oriented database may contain classes which are
nothing but a template of an object. Classes are used to define object and does not store any
data or methods in it however the data and methods are stored in objects. The class is used to
instantiate the object. Classes may be used in object databases to recreate parts of the object
that may not actually be stored in the database. Methods may not be stored in the database and
may be recreated by using a class. With traditional databases, data manipulated by the
application is transient and data in the database is persisted. In object databases, the application
can manipulate both transient as well as persisted data.

These database systems are referred as In-Memory Columnar Database management systems

Two basic methods are used to store objects by different database vendors.

- Each object has a unique ID and is defined as a subclass of a base class, using
  inheritance to determine attributes.
- Virtual memory mapping is used for object storage and management.

Data transfers are either done on a per object basis or on a per page (normally 4K) basis.

5.8.2 Advantages of Object oriented databases

- The objects do not require re-assembling from their component tables each time they
  are used thereby reducing processing overheads by increasing access speeds e.g. up to
  100 times faster for some applications such as Sun Cattel benchmark
- Paging is reduced
- Versioning is easier
- Navigation through the database is easier and more natural, with objects able to contain
  pointers to other objects within the database
- Reuse reduces development costs
- Concurrency control is simplified by the ability to place a single lock on an entire
  hierarchy
- Better data model as based on the 'real world' instead of the 'flattened' relational model
- Good for applications where the relationships between items in the database carry key
  information e.g. in the student database, we were particularly interested in what students
  studied (i.e. the STUDIES relationship). This is handled very efficiently by navigation.
• Relationships and constraints on objects can be stored in the server application rather than the client application therefore any changes need only be made in one place thus reducing the need for and risks involved in making multiple changes
• Fit in well with client/server and distributed architectures

5.8.3 Disadvantages of Object oriented databases

• Poor for applications where the values of items in the database carry key information e.g. if we had been more interested in student age (e.g. to calculate the mean age) than the courses they study then relational database would clearly be more efficient
• Speed of access may be reduced by late binding which may cause extensive searches through the inheritance hierarchies
• Present lack of standards including the lack of a common query language such as SQL (though OQL on its way?)
• There are as yet no formal semantics for ODBMS. Relational databases can be 'proved' correct by means of set theory and relational calculus
• The simplicity of relational tables is lost Object Oriented Databases
• The object oriented paradigm shift can make the move to ODBMS difficult

PRODUCTS

Some of products available in market which uses combined technologies (Columnar database and in-memory database technologies also known as In-Memory Columnar Database System) are given below

HyPer

• It is a hybrid OLTP&OLAP main memory database system. And it is columnar in order to achieve best possible query execution performance for OLAP applications.

HyPer provides best performance due to Data Clustering & Compression and its design choices.

DATA CLUSTERING & COMPRESSION

HyPer’s compression approach in hybrid OLTP & OLAP column stores is based on the observations that while OLTP workloads frequently modify the dataset, they often follow the
working set assumption: only a small subset of the data is accessed and an even smaller subset of this working set is being modified.

Figure 5.12: In-Memory Columnar DBs HyPer Data Clustering and Compression

Hot/cold clustering is an elegant solution to this problem, as the cold bulk of the data can be stored on huge memory pages while the hot, frequently modified working set remains on regular memory pages that can be replicated inexpensively. The frozen, huge data pages are never modified; if a frozen data object is changed, after all, it is invalidated in the frozen partition and reinserted into the hot working set. Some of the design choices are described below

**Design Choices 1**

HyPer relies on in-memory data management without the ballast of traditional database systems caused by DBMS-controlled page structures and buffer management. The SQL table definitions are transformed into simple vector-based virtual memory representations – which constitutes a column-oriented physical storage scheme.

**Design Choices 2**

The OLAP processing is separated from the mission critical OLTP transaction processing by forking virtual memory snapshots. Thus, no concurrency control mechanisms other than the hardware assisted VM management are needed to separate the two workload classes.
Transactions and queries are specified in SQL and are efficiently compiled into LLVM assembly code. The transactions are specified in an SQL scripting language and registered stored procedures. The query evaluation follows a data-centric paradigm by applying as many operations on a data object as possible in between pipeline breakers.

HyPer’s approach to compression in hybrid OLTP & OLAP column stores is based on the observation that while OLTP workloads frequently modify the dataset, they often follow the working set assumption: only a small subset of the data is accessed and an even smaller subset of this working set is being modified.

**Object Store database**

Object Store is a specialized type of database designed to handle data created by applications that use object-oriented programming techniques. It is inspired by the Static database originally developed at Symbolics. ObjectStore is innovative in its use of the C++ language to make database access transparent. Objects can be created in a database by overloading the operator new(). In this way, one can store C++ objects directly in the database and these persistent objects look and behave just like normal C++ objects. By making use of signals, Object Store traps pointer exceptions and transparently brings objects in from the database. In addition, by use of a technique called swizzling, the database can be accessed from different platforms, with
pages being 'swizzled' as they are brought into memory on page faults to correct big endian versus little endian platform issues as well as virtual function table layout.

Object Store has recently expanded its use beyond the object database market to target use as a database for real-time computing, specifically designed for RFID data management, and as a cache for relational databases

**Objectivity Database**

Objectivity Database allows applications to make standard C++, Java, Python or Smalltalk objects persistent without having to convert the data objects into the rows and columns used by a relational database management system (RDBMS). Objectivity/DB supports the most popular object oriented languages plus SQL/ODBC and XML. It runs on Linux, LynxOS, UNIX and Windows platforms. All of the languages and platforms interoperate, with the Objectivity/DB kernel taking care of compiler and hardware platform differences.

**GemStone**

Gemstone builds on the Smalltalk programming language. GemStone systems serve as mission-critical applications. GemStone frameworks still see some interest for web services and service-oriented architectures.

A recent revival of interest in Smalltalk has occurred as a result of its use to generate Javascript for e-commerce web pages or in web application frameworks such as the Seaside web framework. Systems based on object databases are not as common as those based on ORM or Object-relational mapping frameworks such as TopLink or Hibernate. In the area of web application frameworks, JBoss and BEA Weblogic are somewhat analogous to GemStone.

**DB4O Object Database**

DB4O represents an object-oriented database model. One of its main goals is to provide an easy and native interface to persistence for object oriented programming languages. Development with DB4O database does not require a separate data model creation; the application’s class model defines the structure of the data in DB4O database. DB4O attempts to avoid the object/relational impedance mismatch by eliminating the relational layer from a software project. For more information see db4o features.

Developers using relational databases can also benefit from using DB4O, which can be viewed as a complementary tool. The DB4O -RDBMS data exchange can be implemented using DB4O
Replication System (dRS). dRS can also be used for migration between object (DB4O) and relational (RDBMS) technologies.

As an embedded database db4o can be run in application process. It is distributed as a library (jar/dll).

5.8.4 Conclusion

There has been always a requirement of a database management system which are faster in querying and analyzing data for business purpose. As the size of databases are increasing day by day, the query execution and performance of these databases is getting slower and slower or the maintenance cost is going higher and higher.

To overcome these issues, we can take advantages of two technologies called column based database and Main Memory Database management systems. This way we can achieve a database management system having 30-50% higher performance in comparison of DRDB and RODBs.

5.9 Column Based NoSQL DBMS, their Scope and the Future

I extensively researched and published research paper along with my guide Dr. Raghav Mehra and co scholar Dr Nirmal Lodhi on Column Based NoSQL Database, Scope and Future. Paper was published in Dec 2015 in INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS E ISSN 2348 –1269, PRINT ISSN 2349-5138 - [VOLUME 2 I ISSUE 4 I OCT. – DEC. 2015].

Column-oriented or column based database systems, also known as column-stores, have an important demand in the past few years. Basically, it is about storing each database column separately so that the attributes belonging to the same column would be stored contiguously, compressed and densely-packed in the disk. This method has advantages in reading the records faster as compared to classical row stores in which every row are stored one after another in the disk. These databases are more suitable for data warehousing system to get analysis done faster as data is stored in columnar form. Indexes are much faster in column oriented databases which results in faster data retrieval and hence data analysis. This is an alternate database technology over row oriented database systems. To further improve the read performance of system, there are different technologies based on which new DBMS has been designed and
build up, discussed in details in the paper. I would detail only few of the NoSQL DBMS which are available in market.

5.9.1 Column-oriented Database:

A column-oriented DBMS is a DBMS that stores data tables as sections of columns of data rather than as rows of data. In comparison, most relational DBMSs store data in rows. This column-oriented DBMS has advantages for data warehouses, clinical data analysis, CRM systems, and library card catalogs, and other ad hoc inquiry systems where aggregates are computed over large numbers of similar data items.

It is possible to achieve some of the benefits of column-oriented and row-oriented organization with any DBMSs. Denoting one as column-oriented refers to both the ease of expression of a column-oriented structure and the focus on optimizations for column-oriented workloads. This approach is in contrast to row-oriented or row store databases and with correlation databases, which use a value-based storage structure.

Column-oriented storage is closely related to database normalization due to the way it restricts the database schema design. However, it was often found to be too restrictive in practice, and thus many column-oriented databases such as Google's BigTable do allow column groups to avoid frequently needed joins.

A column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. For our example table, the data would be stored in this fashion:

10:001,12:002,11:003,22:004;
Smith:001, Jones:002, Johnson:003, Jones:004;
Joe:001, Mary:002, Cathy:003, Bob:004;
40000:001, 50000:002, 44000:003, 55000:004;

Here any one of the columns more closely matches the structure of an index in a row-based system. This may cause confusion that can lead to the mistaken belief a column-oriented store is really just a row-store with an index on every column. However, it is the mapping of the data that differs dramatically. In a row-oriented indexed system, the primary key is the rowid that is mapped to indexed data. In the column-oriented system, the primary key is the data, mapping back to rowids. This may seem subtle, but the difference can be seen in this common modification to the same store:
…; Smith:001;Jones:002,004;Johnson:003;…

As two of the records store the same value, "Jones", it is possible to store this only once in the column store, along with pointers to all of the rows that match it. For many common searches, like "find all the people with the last name Jones", the answer is retrieved in a single operation. Other operations, like counting the number of matching records or performing math over a set of data, can be greatly improved through this organization.

Whether or not a column-oriented system will be more efficient in operation depends heavily on the workload being automated. It would appear that operations that retrieve data for objects would be slower, requiring numerous disk operations to collect data from multiple columns to build up the record. However, these whole-row operations are generally rare. In the majority of cases, only a limited subset of data is retrieved. In a roledex application, for instance, operations collecting the first and last names from many rows in order to build a list of contacts is far more common than operations reading the data for any single address. This is even truer for writing data into the database, especially if the data tends to be "sparse" with many optional columns. For this reason, column stores have demonstrated excellent real-world performance in spite of many theoretical disadvantages.

This is a simplification. Moreover, partitioning, indexing, caching, views, OLAP cubes, and transactional systems such as write ahead logging or multiversion concurrency control all dramatically affect the physical organization of either system. That said, online transaction processing (OLTP)-focused RDBMS systems are more row-oriented, while online analytical processing (OLAP)-focused systems are a balance of row-oriented and column-oriented.

5.9.2 History of Column-oriented Database

A relational database management system provides data that represents a two-dimensional table, of columns and rows. For example, a database might have this table:

<table>
<thead>
<tr>
<th>RowId</th>
<th>EmpId</th>
<th>Lastname</th>
<th>Firstname</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.00</td>
<td>Smith</td>
<td>Joe</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>12.00</td>
<td>Jones</td>
<td>Mary</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>11.00</td>
<td>Johnson</td>
<td>Cathy</td>
<td>44000</td>
</tr>
<tr>
<td>4</td>
<td>22.00</td>
<td>Jones</td>
<td>Bob</td>
<td>55000</td>
</tr>
</tbody>
</table>
Table 5.5: A Sample table in database

This simple table includes an employee identifier (EmpId), name fields (Lastname and Firstname) and a salary (Salary). This two-dimensional format exists only in theory. In practice, storage hardware requires the data to be serialized into one form or another.

The most expensive operations involving hard disks are seeks. In order to improve overall performance, related data should be stored in a fashion to minimize the number of seeks. This is known as locality of reference, and the basic concept appears in a number of different contexts. Hard disks are organized into a series of blocks of a fixed size, typically enough to store several rows of the table. By organizing the data so rows fit within the blocks, and related rows are grouped together, the number of blocks that need to be read or sought is minimized.

5.9.3 Advantages of Column-oriented DBMS

Comparisons between row-oriented and column-oriented data layouts are typically concerned with the efficiency of hard-disk access for a given workload, as seek time is incredibly long compared to the other delays in computers. Sometimes, reading a megabyte of sequentially stored data takes no more time than one random access. Further, because seek time is improving much more slowly than CPU power this focus will likely continue on systems that rely on hard disks for storage. Following is a set of oversimplified observations which attempt to paint a picture of the trade-offs between column- and row-oriented organizations. Unless, of course, the application can be reasonably assured to fit most/all data into memory, in which case huge optimizations are available from in-memory database systems.

1. Column-oriented organizations are more efficient when an aggregate needs to be computed over many rows but only for a notably smaller subset of all columns of data, because reading that smaller subset of data can be faster than reading all data.

2. Column-oriented organizations are more efficient when new values of a column are supplied for all rows at once, because that column data can be written efficiently and replace old column data without touching any other columns for the rows.

3. Row-oriented organizations are more efficient when many columns of a single row are required at the same time, and when row-size is relatively small, as the entire row can be retrieved with a single disk seek.
4. Row-oriented organizations are more efficient when writing a new row if all of the row data is supplied at the same time, as the entire row can be written with a single disk seek.

In practice, row-oriented storage layouts are well-suited for OLTP-like workloads which are more heavily loaded with interactive transactions. Column-oriented storage layouts are well-suited for OLAP-like workloads such as data warehouses which typically involve a smaller number of highly complex queries over all data.

5.9.4 What is No SQL Database

NoSQL encompasses a wide variety of different database technologies that were developed in response to a rise in the volume of data stored about users, objects and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.

5.9.5 Background of NoSQL Database

The term NoSQL was used by Carlo Strozzi in 1998 to name his lightweight, Strozzi NoSQL open-source relational database that did not expose the standard SQL interface, but was still relational. His NoSQL RDBMS is distinct from the circa-2009 general concept of NoSQL databases. Strozzi suggests that, as the current NoSQL movement departs from the relational model altogether; it should therefore have been called more appropriately NoREL(No Relational').

Johan Oskarsson of Last.fm reintroduced the term NoSQL in early 2009 when he organized an event to discuss "open source distributed, non-relational databases. The name attempted to label the emergence of an increasing number of non-relational, distributed data stores, including open source clones of Google's BigTable/MapReduce and Amazon's Dynamo. Most of the early NoSQL systems did not attempt to provide atomicity, consistency, isolation and durability guarantees, contrary to the prevailing practice among relational database systems.

5.9.6 Most Popular NoSQL Database in Market

The most popular NoSQL databases in market are
1) Mongo DB
2) Apache Cassandra
3) H Base
4) Couch base
5) Solr
6) Elastic Search
7) Splunk
8) Memcached

5.9.7 Types of NoSQL Database

There have been various approaches to classify NoSQL databases, each with different categories and subcategories, some of which overlap. A basic classification based on data model, with examples:

- **Document databases** pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

- **Graph stores** are used to store information about networks, such as social connections. Graph stores include Neo4J and HyperGraphDB.

- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value. Examples of key-value stores are Riak and Voldemort. Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.

- **Wide-column stores** such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

A more detailed classification is the following, based on one from Stephen Yen

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Value Cache</td>
<td>Coherence, eXtreme Scale, GigaSpaces, GemFire, Hazelcast, Infinispan, JBoss Cache, Memcached, Repcached, Terracotta, Velocity</td>
</tr>
<tr>
<td>Key-Value Store</td>
<td>Flare, Keyspace, RAMCloud, SchemaFree, Hyperdex, Aerospike</td>
</tr>
<tr>
<td>Key-Value Store (Ordered)</td>
<td>Actord, FoundationDB, Lightcloud, LMDB, Luxio, MemcacheDB, NMDB, Scalaris, TokyoTyrant</td>
</tr>
<tr>
<td>Tuple Store</td>
<td>Apache River, Coord, GigaSpaces</td>
</tr>
<tr>
<td>Object Database</td>
<td>DB4O, Objectivity/DB, Perst, Shoal, ZopeDB</td>
</tr>
<tr>
<td>Document Store</td>
<td>Clusterpoint, Couchbase, CouchDB, DocumentDB, Lotus Notes, MarkLogic, Mongo DB, Qizx, XML-databases</td>
</tr>
<tr>
<td>Wide Store</td>
<td>Columnar Store</td>
</tr>
</tbody>
</table>

Table 5-6: database types with example

5.9.8 Advantages of NoSQL Database

When compared to relational databases, NoSQL databases are more scalable and provide superior performance and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of structured, semi-structured, and unstructured data
- Agile sprints, quick iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Efficient, scale-out architecture instead of expensive, monolithic architecture

5.9.9 Trending & Benchmarking NoSQL Databases: Cassandra vs. MongoDB vs. HBase vs. Couchbase

Understanding the performance behavior of a NoSQL database like Apache Cassandra™ under various conditions is critical. Conducting a formal proof of concept (POC) in the environment in which the database will run is the best way to evaluate platforms. POC processes that include the right benchmarks such as production configurations, parameters and anticipated data and concurrent user workloads give both IT and business stakeholder’s powerful insight about platforms under consideration and a view for how business applications will perform in production.

Independent benchmark analyses and testing of various NoSQL platforms under big data, production-level workloads have been performed over the years and have consistently identified Apache Cassandra as the platform of choice for businesses interested in adopting NoSQL as the database for modern Web, mobile and IOT applications. One benchmark analysis (Solving Big Data Challenges for Enterprise Application Performance Management) by engineers at the University of Toronto, which in evaluating six different data stores, found Apache Cassandra the “clear winner throughout our experiments”. Also, End Point Corporation, a database and open source consulting company, benchmarked the top NoSQL
databases including: Apache Cassandra, Apache HBase, Couchbase, and MongoDB using a variety of different workloads on AWS EC2. The databases involved were:

**Apache Cassandra:** Highly scalable, high performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

**Apache HBase:** Open source, non-relational, distributed database modeled after Google’s BigTable and is written in Java. It is developed as part of Apache Software Foundation’s Apache Hadoop project and runs on top of HDFS (Hadoop Distributed File System), providing BigTable-like capabilities for Hadoop.

**MongoDB:** Cross-platform document-oriented database system that eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas making the integration of data in certain types of applications easier and faster.

**Couchbase:** Distributed NoSQL document-oriented database that is optimized for interactive applications.

End Point conducted the benchmark of these NoSQL database options on Amazon Web Services EC2 instances, which is an industry-standard platform for hosting horizontally scalable services. In order to minimize the effect of AWS CPU and I/O variability, End Point performed each test 3 times on 3 different days. New EC2 instances were used for each test run to further reduce the impact of any “lame instance” or “noisy neighbor” effects sometimes experienced in cloud environments, on any one test.

**5.9.10 NoSQL Database Performance Testing Results**

When it comes to performance, it should be noted that there is (to date) no single “winner takes all” among the top NoSQL databases or any other NoSQL engine for that matter. Depending on the use case and deployment conditions, it is almost always possible for one NoSQL database to outperform another and yet lag its competitor when the rules of engagement change. Here are a couple snapshots of the performance benchmark to give you a sense of how each NoSQL database stacks up.
Throughput by Workload

Each workload appears below with the throughput/operations-per-second (more is better) graphed vertically, the number of nodes used for the workload displayed horizontally, and a table with the result numbers following each graph.

Load process

For load, Couchbase, HBase, and Mongo DB all had to be configured for non-durable writes to complete in a reasonable amount of time, with Cassandra being the only database performing durable write operations. Therefore, the numbers below for Couchbase, HBase, and MongoDB represent non-durable write metrics.

![Bar chart showing throughput per workload and nodes for Cassandra, Couchbase, HBase and MongoDB.](image)

Figure 5.14: Load performance of Cassandra, MongoDB, HBase and Couchbase

The table below has the number of operations processed by mongodh, Cassandra, couvhbase and Hbase. We observed that the transaction processed by Casandra are way higher than that of rest of the databases technologies.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Cassandra</th>
<th>Couchbase</th>
<th>HBase</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20K</td>
<td>10K</td>
<td>15K</td>
<td>25K</td>
</tr>
<tr>
<td>2</td>
<td>40K</td>
<td>20K</td>
<td>30K</td>
<td>50K</td>
</tr>
<tr>
<td>4</td>
<td>80K</td>
<td>40K</td>
<td>60K</td>
<td>100K</td>
</tr>
<tr>
<td>8</td>
<td>160K</td>
<td>80K</td>
<td>120K</td>
<td>200K</td>
</tr>
<tr>
<td>16</td>
<td>320K</td>
<td>160K</td>
<td>240K</td>
<td>400K</td>
</tr>
<tr>
<td>32</td>
<td>640K</td>
<td>320K</td>
<td>480K</td>
<td>800K</td>
</tr>
<tr>
<td>1</td>
<td>18,683.43</td>
<td>13,761.12</td>
<td>15,617.98</td>
<td>8,368.44</td>
</tr>
<tr>
<td>----</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>2</td>
<td>31,144.24</td>
<td>26,140.82</td>
<td>23,373.93</td>
<td>13,462.51</td>
</tr>
<tr>
<td>4</td>
<td>53,067.62</td>
<td>40,063.34</td>
<td>38,991.82</td>
<td>18,038.49</td>
</tr>
<tr>
<td>8</td>
<td>86,924.94</td>
<td>76,504.40</td>
<td>74,405.64</td>
<td>34,305.30</td>
</tr>
<tr>
<td>16</td>
<td>173,001.20</td>
<td>131,887.99</td>
<td>143,553.41</td>
<td>73,335.62</td>
</tr>
<tr>
<td>32</td>
<td>326,427.07</td>
<td>192,204.94</td>
<td>296,857.36</td>
<td>134,968.87</td>
</tr>
</tbody>
</table>

Table 5-7: Operations per second in of Cassandra, MongoDB, HBase and Couchbase

5.9.11 Mixed Operational and Analytical Workload

Note that Couchbase was eliminated from this test because it does not support scan operations (producing the error: “Range scan is not supported”).

![Mixed Load (operational and Analytical) performance of Cassandra, HBase and MongoDB](image)

Figure 5.15: Mixed Load (operational and Analytical) performance of Cassandra, HBase and MongoDB
<table>
<thead>
<tr>
<th></th>
<th>Operations per second in of Cassandra, MongoDB, HBase and Couchbase during mixed load</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10,386.08</td>
</tr>
<tr>
<td>4</td>
<td>18,720.50</td>
</tr>
<tr>
<td>8</td>
<td>36,773.58</td>
</tr>
<tr>
<td>16</td>
<td>78,894.24</td>
</tr>
<tr>
<td>32</td>
<td>128,994.91</td>
</tr>
</tbody>
</table>

5.9.12 NoSQL Database Performance

These performance metrics are just a few of the many that have solidified Apache Cassandra as the NoSQL database of choice for businesses needing a modern, distributed database for their Web, mobile and IOT applications. Each database option (Cassandra, HBase, Couchbase and MongoDB) will certainly shine in particular use cases, so it’s important to test your specific use cases to ensure your selected database meets your performance SLA. Whether you are primarily concerned with throughput or latency, or more interested in the architectural benefits such as having no single point of failure or being able to have elastic scalability across multiple data centers and the cloud, much of an application’s success comes down to its ability to deliver the response times Web, mobile and IOT customers expect. As the benchmarks referenced here showcase, Cassandra’s reputation for fast write and read performance, and delivering true linear scale performance in a master-less, scale-out design, bests its top NoSQL database rivals in many use cases.

5.9.13 Scope and Future of NoSQL Database

The global NoSQL market is forecast to reach $3.4 Billion in 2020, representing a compound annual growth rate (CAGR) of 21% for the period 2015 – 2020.

The fledgling NoSQL marketplace is going through a rapid transition – from the predominantly community-driven platform development to a more mature application-driven market. Scaling up web infrastructure on NoSQL basis have proven successful for Facebook, Digg and Twitter. Successful attempts have been made to develop NOSQL applications in the biotechnology, defense and image/signal processing. Interest in using key-value pair (KVP) technology has reemerged to the point where the traditional RDMS vendors evaluate strategy of developing
in-house NoSQL solutions and integrating them in current product offers. It will not take long before we’ll see acquisitions driven by emerging NoSQL technology. The future deals will likely be made to better compete both in platform offering and in vertical market segments.

The worldwide NoSQL market is posed for two digit growth in the period 2015-2020. NoSQL is moving in to become a major player in database marketplace.

5.9.14 Conclusion

NoSQL is still relatively new, and while some have adopted it wholeheartedly, others range from reluctant to completely oppose to the new technology. There are a number of challenges that NoSQL still faces:

Lack of trained administrators and developers. The most senior developers and admins have made their livelihood writing code for and managing SQL databases. NoSQL may be uncharted territory for many. Many CIOs are cautious and avoid jumping into new technology that they feel might not be ready for primetime. While some of the NoSQL databases might have even reached maturity, many CIOs want to see more analytic and transactional applications built around NoSQL and Big Data technology. They also want application development tools that allow them to build their own custom apps. The future mostly looks bright for NoSQL, and it is clear that it has found a place for itself in the Big Data market alongside Hadoop. Over the next few years, expect to see more competing open source implementations, more startups, and more corporations partnering with startups and/or acquiring them. This will lead to more developers and administrators trained in NoSQL databases and ultimately more customer adoption.

5.10 Get best performance for analytics database systems by merging “in-memory” and “column oriented database” technologies

I extensively researched and published research paper under the assistance of my guide Dr. Raghav Mehra on To get best performance for analytics database systems by merging “in memory database” and “column oriented database”. Paper was published in March 2016 in INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS(IJRAR) e ISSN 2348 –1269, Print ISSN 2349-5138 [VOLUME 3 I ISSUE 1 I JAN. – MARCH 2016].
The research describes a unique way to combine technologies to get better performance for a database management system.

In order to get best performance for an analytic system or data warehouse systems, two technologies, column oriented database management systems and main memory database management system can be combined to get advantages of these two. Both technologies give its best performance to opponent database system, for example Main memory database management systems are faster due the fact that these reside in main physical memory as compared to disk resident database systems. This is because RAM is faster than hard drive/disk. The main memory databases systems are100 times faster than that of disk resident databases. The column oriented databases are faster than the row based database because in column oriented databases, data is stored in columns and indexed as compared to row based database systems. Similarly column based database systems are 100 times faster than that of row based database systems By combining these two technologies, we can achieve 200 times better performance for analytics systems they needs to be high performing in analysis and computation.

5.10.1 DEFINITION
5.10.2 Analytical database Systems

Analytic database or Analytical database is a database system which stores historical read only data on business dimensions such for example revenue for each quarter each year or profit of organization quarter on quarter or year on year etc. Database can be queried by business users which have access to generate and analyze report based on their own custom requirements or they can generate pre-defined reports based on the SQL statement they have to generate reports. These databases are a component of data warehouse systems or data mart, specially designed to support decision support system such as business intelligence and analytical application. These are different from online transactional database and have better read and analytical performance while executing a report.

There are five analytical database solutions available

i. Column Based Database Systems
ii. Main Memory Database Systems
iii. Data warehouse Applications
iv. Massively Parallel Processing Database Systems
v. Online Analytical Processing Database Systems

5.10.3 Main Memory Database

Main memory database system (MMDB) also called real-time database (RTDB) or in memory database (IMDB) is a database management system that reside in main memory rather disk. Since physical memory is considerably faster than a hard disk or a solid state disk, which are faster than hard disk, complex systems which applies to a disk storage mechanism. In-memory databases are faster than disk-optimized databases due to the internal optimization algorithms are simpler and execute lesser CPU instructions. Decision support queries can be satisfied much more rapidly and high-end computers can be configured with terabytes of memory. Due to faster speed of RAM, In-Memory Database are way faster in comparison of disk storage database systems it is contrasted with database management

Accessing in memory data reduces the Disk seek while reading the data which provides quicker and better predictable read performance in comparison that of disk.

5.10.4 Architecture

A sample architecture of main memory database system is shown in Figure below. It depicts a main memory database management system. This has nearly all the components except interface to connect to difference programs such as ODBC or JDBC drivers, which are present in disk resident database management system. Implementations of components under SQL Engine, Relational Engine, and Storage Engine differ heavily from the DRDB components.
5.10.5 Advantages of In Memory Database Management Systems over Disk Storage Database Systems
1. Another variation is to have huge amounts of nonvolatile memory on hosts/servers. E.g. Flash memory chips as addressable memory rather than structured as disk arrays. Databases in this form of memory combines very quick access speed with persistence over reboots and power losses.

2. In-memory databases are often used for applications which demands high response time is, such as mobile advertising networks and telecommunications network equipment.

3. IMDBs have gained a huge traction, predominantly in the data analytics, beginning mid-2000s mainly due to cheaper RAMs.

4. Apart from providing extremely fast data read / response times, in-memory analytics eliminate the need of data indexing and storing of pre-aggregated data in OLAP cubes or aggregate tables. This capacity minimizes IT costs and enables faster implementation of BI and BA applications.

5. The developments in recent past have made in-memory analytics database increasingly possible: 64-bit computing, multicore hosts and cheaper RAM prices

6. Main memory databases store data on volatile memory devices. These devices lose all stored information due to power off or is reset. In this case, MMDBs can be said to lack support for the durability of the ACID (atomicity, consistency, isolation, durability) properties. The Volatile memory-based MMDBs can, and often do, support the atomicity, consistency and isolation of ACID properties.

5.10.6 Column Oriented Database

A Database Management Systems which stores its contents/data in columns instead of rows. This is all about data store technology, here database column separately so that the attributes belonging to the same column would be stored contiguously, compressed and packed densely in the disk. The advantage of this technology is in reading the records faster as compared to classical row-stores where in rows are stored one after another in the disk.

These databases are more beneficial for analytics systems to get analysis done faster as data is stored in columnar form. Indexes are much faster in column oriented databases hence the faster data retrieval for data analysis. This is an alternate database technology over row oriented databases.
There are two technologies to map database tables onto a one-dimensional interface: store the table row-by-row or store the table column-by-column. The row-by-row technology keeps all information about an entity together.

Take an example of author table

**Author Table**

<table>
<thead>
<tr>
<th>Author ID</th>
<th>Author Name</th>
<th>Age</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ram Singh</td>
<td>30</td>
<td>Pune</td>
</tr>
<tr>
<td>2</td>
<td>Rudra Singh</td>
<td>4</td>
<td>Delhi</td>
</tr>
<tr>
<td>3</td>
<td>Riddhima Singh</td>
<td>3</td>
<td>Faridabad</td>
</tr>
<tr>
<td>4</td>
<td>Nirmal Singh</td>
<td>30</td>
<td>Faridabad</td>
</tr>
</tbody>
</table>

**Table 5-9: Sample Author table**

**Row by row storage technology**

Author table will stored as below using row-by-row methodology or technology storage

1, Ram Singh, 30, Pune; 2, Rudra Singh, 4, Pune; 3, Riddhima Singh, 3, Faridabad; 4, Nirmal Singh, 30, Faridabad;

**Column -by-column storage technology**

Author table will stored as below using row-by-row methodology or technology storage

1, 2, 3, 4; Ram Singh, Rudra Singh, Riddhima Singh, Nirmal Lodhi; 30,4,3,30, Pune, Delhi, Faridabad, Faridabad;

In the author example above, it will store all information about the first Author, and then all information about the second Author, etc.

The column-by-column approach keeps all attribute information together: the entire author id will be stored consecutively, then all of the author names, et cetera. Both approaches are reasonable designs and typically a choice is made based on performance expectations. If the
expected work load tends to access data on the granularity of an entity, then the row-by-row storage is preferable since needed information will be stored together.

5.10.7 1.3.1 Advantages of Column Oriented Database Management Systems over Row Oriented Database Systems

1. High performance on aggregation queries (like COUNT, SUM, AVG, MIN, MAX)
2. Highly efficient data compression and/or data partitioning
3. True scalability and fast data loading for Big Data
4. Accessible by many 3rd party BI analytic tools
5. Fairly simple systems administration

Data warehouses Applications, Massively Parallel Processing Database Systems, online Analytical Processing Database Systems are out of scope of this article.

5.10.8 PRODUCTS

Couple of products available in market is give below

SAP HANA

HANA DB takes advantage of the low cost of RAM, data processing abilities of multicore CPUs and fast data retrieval from solid-state drives relative to traditional hard drives to provide better performance of analytical and transactional systems. It enables a multiengine query processing environment that allows it to support relational databases (with both row-oriented and column-oriented physical representations within a hybrid engine) as well as text and graph processing for semi and unstructured database systems in the same system. HANA follows 100% ACID properties. While HANA has been called variously an acronym for HAsso's New Architecture (this is a reference to SAP founder Hasso Plattner) and High Performance Analytic Appliances, HANA is a name not an acronym.

EXASolution

EXASolution is a parallelized relational DBMS runs on a cluster of standard hardware hosts. Following the SPMD model, identical code is executed simultaneously on every node. The data is stored in a columnar way and proprietary In-Memory compression methods are used. Company claims that tuning efforts are not required as the database includes self-optimization (like automatic indices, statistics, and distributing of data). EXASolution is designed to run In
Memory, although data is stored persistently on disc following the ACID rules. EXASolution supports the SQL Standard 2003 and can be integrated through standard interfaces like JDBC, ODBC or ADO.NET. Additionally, a SDK is provided for native integration. License model is based on the RAM allocated for the database software (per GB RAM) and independent to the physical infrastructure. Customers get the maximum performance if compressed active data fits into that licensed RAM, but it can also be huge.

EXASOL implemented a Cluster Operating System (EXACluster OS). It is based on standard Linux and have supports parallelism functionality. It can be compared to Virtualization, but instead of virtualizing the hardware of a server, it virtualizes a cluster of nodes to a single node. The Cluster management algorithms are provided such as failover mechanisms or auto cluster installation.

5.10.9 CONCLUSION

There has been always a requirement of a databases which provides faster query execution and analysis for business purpose. As the size of databases are increasing day by day, the query execution and performance of these databases is getting slower and slower or the maintenance cost is going higher and higher. To overcome these issues, we can take advantages of two technologies called column based database and Main Memory Database management systems. This way we can achieve a database management system having 200 times higher performance in comparison of DRDB / RODBs. This has been achieved by Oracle Corporation by implementing these two technologies in oracle 12 c on EXADATA platform is just an example to show the improved performance.

5.11 Performance Management of In Memory Databases

In-memory databases are becoming increasingly popular and an ever-more important factor in performance-critical activities such as stream processing and deep data analytics. Join Julian Stuhler as he delves into the world of in-memory databases: a technology that’s both reassuringly familiar and intriguingly novel at the same time. In memory database systems are fastest database systems as the database itself resides in system’s memory, so whenever there is any query comes to it need not to go to hard disk to read or write data to hard dis saving time and performance of the entire system.
Google, Twitter, Facebook and many others all rely on various forms of in-memory database to provide rapid response times in the face of ever increasing data volumes.

5.11.1 WHAT IS IN MEMORY DATABASE?

An in-memory database (IMDB) or main memory database system (MMDB) or real-time database (RTDB) is a database management system that primarily relies on main memory for computer data storage. It is contrasted with database management systems which employ a disk storage mechanism. Main memory databases are faster than disk-optimized databases since the internal optimization algorithms are simpler and execute fewer CPU instructions.

Accessing data in memory reduces the Disk seek when querying the data which provides faster and more predictable performance than disk.

5.11.2 EVOLUTION OF IN MEMORY DATABASES

I first bumped into the in-memory database concept back around 2001, and since then have considered it a classic disruptive technology that would inevitably marginalize, then unseat conventional SQL databases.

All the old R-series derived database optimization strategies from back in the 1970's operate in a linear fashion, like a zipper. They assume you read one record at a time from each table ("stream"), compare those records with each other ("join"), and immediately write the result back to disk.

The old database design assumed you never have more than one record from each table in memory at a time, because who has that much memory? Everything is centered on that design assumption: you have a tiny amount of ram, so everything comes from disk in bite sized pieces and goes right back again. This assumption was undermined by Moore's Law, which doubled the available ram every year for the past few decades, and continues to do so.

In 2004, the department of defense commissioned a 2.5 terabyte ram disk for use as a database server. The industry switched to 64 bit (x86-64) processors in 2005. The Tyan Thunder supported 64 gigabytes of ram, the Nvidia MCP55 pro chipset supported 128 gigs, the elegantly named "Tyan S4989WG2NR-SI" supported 256 gigs, and these days the super micro 7500 supports half a terabyte of RAM.
So these days 64-bit systems with half a terabyte of ram are available retail. (And if you want to wander away from the PC, IBM PowerPC systems like the 780 server can hold a couple terabytes RAM each.) Keeping all your tables in memory means you literally get a 3 orders of magnitude speedup (1000x), and you can use simple generic indexing strategies so the code becomes really simple. The first entirely in-memory database implementation I saw (a source forge project back in 2001) was a 1000 line python implementation on source forge that stored everything in python dictionaries. Over 90% of the code of that was implementing the SQL parser, not the actual database.

The more recent "nosql" movement is basically following up on this by saying "ok, if we've got our tables in memory why do we need SQL to talk to them?" The obvious way to design an in memory database is thus in two parts: a direct layer providing function access to the data store with transactions and searching and persistence and such, and the other providing an SQL layer on top of that. Whether the database is in shared memory or not is an implementation detail.

The current crop of databases (mysql, postgresql, oracle, and so on) is all painfully obsolete. As with the introduction of UNIX, they were obsoleted by something much simpler, which figured out that 90% of the stuff they spent their time doing didn't need to be done at all.

5.11.3 MECHANISMS

Many MMDBs add durability via the following mechanisms: Snapshot files, or, checkpoint images, which record the state of the database at a given moment in time. These are typically generated periodically, or, at least when the MMDB does a controlled shut-down. While they give a measure of persistence to the data (in that not everything is lost in the case of a system crash) they only offer partial durability (as 'recent' changes will be lost). For full durability, they will need to be supplemented by one of the following:

Transaction logging, which records changes to the database in a journal file and facilitates automatic recovery of an in-memory database High availability implementations that rely on database replication, with automatic failover to an identical standby database in the event of primary database failure To protect against loss of data in the case of a complete system crash, replication of a MMDB is normally used in conjunction with one or more of the mechanisms listed above.
Some MMDBs allow the database schema to specify different durability requirements for selected areas of the database - thus, faster changing data that can easily be regenerated or that has no meaning after a system shut-down would not need to be journal for durability (though it would have to be replicated for high availability), whereas configuration information would be flagged as needing preservation.

5.11.4 ARCHITECTURE

![Diagram of Main Memory Database System]

Figure 5.17: Sample Architecture of Main Memory Database System with driver interface
Figure, depicts a main memory database management system. It has the interface to interact with JDBC or ODBC drivers in order to have transaction with external world (applications). This has nearly all the components, which are present in disk resident database management system. Implementations of components under SQL Engine, Relational Engine, and Storage Engine differ heavily from the DRDB components.

5.11.5 HYBRIDS WITH ON-DISK DATABASES

The first database engine to support both in memory and on-disk tables in a single database were released in 2003. The advantage to this approach is flexibility: the developer can strike a balance between performance (which is enhanced by sorting, storing and retrieving specified data entirely in memory, rather than going to disk); cost, because a less costly hard disk can be substituted for more memory; persistence; and form factor, because RAM chips cannot approach the density of a small hard drive. Manufacturing efficiency is another reason a combined in-memory/on-disk database system may be chosen. Some device product lines, especially in consumer electronics, include some units with permanent storage, and others that rely on memory for storage (set-top boxes, for example). If such devices require a database system, a manufacturer can adopt a hybrid database system at lower and upper cost, and with less code customization, than using separate in-memory and on-disk databases, respectively, for its disk-less and disk-based products.

5.11.6 STORAGE MEMORY

Another variation is to have large amounts of nonvolatile memory in the server. For example flash memory chips as addressable memory rather than structured as disk arrays. A database in this form of memory combines very fast access speed with persistence over reboots and power losses.

5.11.7 IMPORTANCE OF IN MEMORY DATABASE

In applications where response time is critical, such as telecommunications network equipment and mobile advertising networks, main memory databases are often used. IMDBs have gained a lot of traction, especially in the Data analytics space, starting mid-2000s mainly due to cheaper RAMs.
In addition to providing extremely fast query response times, in-memory analytics can reduce or eliminate the need for data indexing and storing pre-aggregated data in OLAP cubes or aggregate tables. This capacity reduces IT costs and allows faster implementation of BI/BA applications.

Three developments in recent years have made in-memory analytics increasingly feasible: 64-bit computing, multi-core servers and lower RAM prices main memory databases store data on volatile memory devices. These devices lose all stored information when the device loses power or is reset. In this case, MMDBs can be said to lack support for the durability portion of the ACID (atomicity, consistency, isolation, durability) properties. Volatile memory-based MMDBs can, and often do, support the other three ACID properties of atomicity, consistency and isolation.

5.11.8 PRODUCTS

I first bumped into the in-memory database concept back around 2001, and since then have considered it a classic disruptive technology that would inevitably marginalize, then unseat conventional SQL databases In actual, development of a relational in memory database system was started at Perihelion Software in 1991, and had its first commercial release in early 1993 under the name Polyhedra. The product was later spun out as a separate company, which was acquired by Enea AB in 2001. Polyhedra were developed from the start as a commercial offering for use in SCADA and embedded systems.

Companies needing a fast data storage mechanism for their own products have often developed their own, in-house solution which they later marketed commercially. For example, research in main-memory database systems started around 1993 at Bell Labs. It was prototyped as the Dali Main-Memory Storage Manager. This research lead to the commercial main-memory database, Datablitz. In 2001, McObject introduced eXtremeDB, the first in-memory database system targeting embedded systems, with early adoption in sectors including telecom, industrial control and consumer electronics. Later, they added features associated with non-embedded computing, including 64-bit support and SQL/ODBC/JDBC interfaces, and eXtremeDB has seen adoption in enterprise, financial and Web-based systems requiring low latency. In recent years, main memory databases have attracted the interest of larger database vendors.
Ehcache, is an in-memory database created by the developers of Terracotta, Inc.. It can hold the largest amount of data in memory on the smallest number of servers. SAG acquired the San Francisco based company in 2010 to imbed with their BPM solutions. TimesTen, a start-up company founded by Marie-Anne Neimat in 1996 as a spin-off from Hewlett-Packard, was acquired by Oracle Corporation in 2005. Oracle now markets this product as both a standalone database and an in-memory database cache to the Oracle database.

In 1999, Altibase Corporation developed an in-Memory DBMS offering. In 2012, Altibase released version 6 of its hybrid database flagship product, ALTIBASE HDB. Also in 1999, Microsoft COM+ IMDB solution provided an application with fast access to data through OLE DB, without incurring the overhead associated with storing and accessing data to and from physical disks that worked within Windows NT 3.5 and then upcoming Windows 2000. IBM acquired solidDB in 2008, and Microsoft is widely rumored to be launching an in-memory solution in 2009. VoltDB, founded by DBMS pioneer Michael Stonebraker, announced the general availability of its namesake in-memory database in May 2010, and offers versions of the product under open source (GPLv3) and commercial licenses. SAP AG announced general availability of their own in-memory product, SAP HANA, in June 2011. Oracle also has a similar product to SAP's called Oracle Exalytics. WebDNA 7, released as freeware, is a robust hybrid in-memory database system and scripting language designed for the World Wide Web.

5.11.9 Conclusion

In applications where response time is critical, such as telecommunications network equipment and mobile advertising networks, main memory databases are often used. IMDBs have gained a lot of traction, especially in the Data analytics space, starting mid-2000s mainly due to cheaper RAMs. In addition to providing extremely fast query response times, in-memory analytics can reduce or eliminate the need for data indexing and storing pre-aggregated data in OLAP cubes or aggregate tables. This capacity reduces IT costs and allows faster implementation of BI/BA applications.

Three developments in recent years have made in-memory analytics increasingly feasible: 64-bit computing, multi-core servers and lower RAM prices Main memory databases store data on volatile memory devices. These devices lose all stored information when the device loses power or is reset. In this case, MMDBs can be said to lack support for the durability portion of the ACID (atomicity, consistency, isolation, durability) properties. Volatile memory-based
MMDBs can, and often do, support the other three ACID properties of atomicity, consistency and isolation.

Finally, no discussion on in-memory databases would be complete without at least a brief mention of the high performance in-memory database systems that have been built from the ground up to support today’s search and social networking sites. Google, Twitter, Facebook and many others all rely on various forms of in-memory database to provide rapid response times in the face of ever increasing data volumes. Other data in memory solutions such as SolidDB (acquired by IBM in 2007) provide a somewhat more generalized solution to address many of the same issues, and can even be used as a kind of high performance front end to more conventional disk based databases. From a simple HDD swap to SSD-aware RDBMS systems and beyond, in-memory databases are becoming increasingly popular and an ever-more important factor in performance-critical activities such as stream processing and deep data analytics.

Expect to hear much more about them over the next few years.

5.12 Conclusion and Future Work

In this thesis, I have shown that buffer management can have a very significant impact on the performance of a priority oriented database system, especially when the unpredictability of the workload forces the system to operate in regions where the total buffer requirements of the concurrent transactions exceed the system’s buffer capacity. We have introduced a new buffer management algorithm, called Priority-Cards, that uses page-level information provided by the database access methods to make priority-based buffer management decisions. The performance of our new algorithm has been compared to the performance of Priority-LRU and Priority-DBMIN. Two algorithms proposed earlier for priority-based buffer management.

A number of performance insights have been obtained as a result of simulation experiments. Priority-Cards was shown to perform better than Priority-LRU for all of the workloads considered here. For most workloads, Priority-Cards performed almost as well as Priority-DBMIN; for some workloads with data sharing, Priority-Cards actually provided better performance than Priority-DBMIN. Even when the workload consisted of transactions of equal priority, Priority-Cards performed significantly better than Priority-LRU and almost as well as Priority-DBMIN.
These results are significant for several reasons: First, in previous studies [Chou85, Care891 it has been shown that DBMIN like approaches to buffer management provide better performance than approaches based on simple strategies such as LRU. Still, most existing database systems continue to use LRU-based approaches because they do not require as much information as DBMIN does. Second, the type of information used by the buffer manager in Priority-Cards is already being provided to buffer managers in existing database systems.

Our algorithm has the advantages of both DBMIN-based approaches and LRU-based approaches; it provides good performance while requiring little information. Finally, we have also shown that Priority-Cards adapts itself dynamically as data sharing increases, while Priority-DBMIN’s more static approach to buffer allocation can cause its performance to suffer in the presence of data sharing. We plan to continue our work in the area of priority-based DBMS scheduling. For example, we will extend our algorithms to workloads consisting of multi-query transactions.

We also intend to study the problem of concurrency control conflicts in a priority-oriented DBMS. Finally, we plan to extend our performance study to the real-time context, where the workloads contain transactions with deadlines and importance levels.