# PROCESSING OF READ-ONLY QUERIES AT A REMOTE BACKUP

Christos A Polyzois
Hector Garcia-Molina

CS-TR-354-91

November 1991

# Processing of Read-Only Queries at a Remote Backup

Christos A. Polyzois and Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544
e-mail: cap@Princeton.EDU

*ABSTRACT*

Remote backup systems are often used to provide high data availability. Updates are typically propagated to the backup via a log, which decouples the backup from the primary. We show that this decoupling can lead to efficient installation of updates in batches and efficient processing of read-only queries, by eliminating or reducing access conflicts between updates and queries. We present several methods for query processing at the backup and evaluate their performance analytically.

## 1. Introduction

A *remote backup* is a copy of a primary database maintained at a geographically separate location. Such a copy can take over transaction processing if the primary copy is unable to continue processing. The geographic separation of the two copies offers protection against extensive failures (*disasters*) for which local replication techniques are inadequate. Possible causes of such failures include environmental hazards (fire, flood, earthquake, power outage), malicious acts and operator errors (some of which can be contained if the copies are administered separately).

A remote backup represents a significant investment: the computing resources employed at the backup are typically comparable to those at the primary; in addition, network resources are required to propagate to the backup the changes installed at the pri-

mary, in order to keep the backup up-to-date with respect to the primary. Some critical applications need the protection offered by a remote backup and are willing to pay its price. However, it would be profitable for these applications to exploit the backup for useful computation as well, rather than just for monitoring the primary. Such use would essentially lower the cost of maintaining the backup, and could possibly make a remote backup affordable for other, less critical applications.

In this paper we examine mechanisms for running read-only queries at the backup. A number of mechanisms have been suggested in the literature for processing read-only queries [1],[3],[6],[8],[9]. These include multi-version algorithms that allow queries to read older snapshots of the database. The disaster recovery scenario also involves multiple copies, one at the primary and one at the backup, and queries will read an older snapshot of the data. However, what distinguishes the disaster recovery scenario from others is the way the updates are installed at the backup copy.

Existing backup systems [2],[7],[12] are invariably *log based*. Update transactions run only at the primary, where they produce a log of the changes they make. The log is propagated to the backup and it is then applied to the backup database copy. Log based backup management *decouples* the installation of updates at the backup from the primary transaction processing and leads to improved performance at the backup. In particular, updates can be applied more efficiently in batches. For instance, a group of updates may be sorted and applied in physical disk order, reducing seek distances.

Query algorithms at the backup can also take advantage of the primary/backup decoupling, so that query processing will not impact primary transaction processing at all. Query processing still interferes with update installation that the backup, but given

the nature of updates, several key optimizations can be performed. For instance, queries executed when no updates are being installed need no concurrency control at all. Queries that run during update installation may know *in advance* what objects will be modified by a batch of updates and can use this knowledge to improve performance. In this paper we study such optimizations, presenting several simple but very efficient algorithms for query processing at the backup. The performance of these algorithms is studied through an analytical model.

Note that batch installation of updates is possible in both *1-safe* and *2-safe* systems [4]. In 2-safe systems, a transaction cannot commit at the primary until its log records are safely received at the backup (not necessarily applied to the backup database). In 1-safe systems, log records are propagated *after* the transaction commits at the primary. (With 1-safe processing resources are held for a shorter time leading to improved throughput at the primary; however, some transactions may be lost in case of a disaster.) In either case, application (or installation) of updates at the backup occurs after the transaction commits at the primary. Thus, the application is decoupled from primary transaction processing.

Note that we have presented log based, decoupled backup copies in the context of disaster recovery. However, this scenario may be justified even when reliability is not the major consideration. In some cases queries may naturally arise at one computer and updates at another. For example, a company may use front-end computers at stores (modifying inventory, recording orders, etc.), and a back-end computer at headquarters for queries (trend analysis, budget summaries, etc.). In other cases where the query load is significant, decoupled query processing may actually provide the most effective solu-

tion. Finally, it may sometimes be desirable to run queries on a computer with different software or a different environment than the one running updates. In all these scenarios, one may obtain best performance with the query processing algorithms we study here, while achieving high reliability.

The rest of the paper is organized as follows: in Section 2 we develop a simplified framework and we present the main idea of our scheme. In Section 3 we attempt a classification of queries and discuss the suitability of our scheme for the various classes of queries. In Section 4 we discuss several methods for processing read-only queries and in Section 5 we evaluate their performance. Section 6 addresses the size of update batches and Section 7 concludes the paper.

## 2. System Framework and Mechanism Overview

To make our presentation easier, we assume a very simple model with just one primary and one backup computer. The primary maintains a log of the transaction processing activity. This log is sent to the backup to enable it to install the same updates that were installed at the primary. The log stream is sometimes decomposed further by the backup, e.g., into separate mini-streams for each disk pack.

The techniques we present under this simple model are also applicable to the general case of systems with multiple primary computers, multiple backup computers and multiple log streams between the primaries and the backups. In such an environment, the epoch algorithm [5] can be used to propagate the logs to the backup computers and ensure that the assumptions we make in the single log stream configuration also hold true for the case of multiple log streams. Due to space limitations, we cannot consider this issue here. For details, see [5].

In our framework a backup computer serves two purposes: it installs the changes it receives in the log stream in order to keep pace with its primary peer and it allows read-only queries to read the data. These two tasks conflict with each other, since the update activity is trying to modify data which read-only queries may need to read. Thus, it is necessary to resolve the conflict between these tasks.

Our approach to resolving this conflict is to install the updates received by the backup computer on its log stream in groups rather than continuously. Periodically, the primary inserts special markers into the log stream. These markers serve as delimiters that denote the end of the current group of transactions and the start of a new one. The backup computer accumulates the logs it receives, but it does not apply the changes immediately. Rather, it waits until it receives the next marker. Then, it applies the updates of all transactions that committed between the last two markers received (i.e., the last group of transactions encountered). In Section 6 we mention the factors that must be considered in selecting the time between markers (i.e., group size). In our simple model, the markers could be placed in the log by the backup rather than the primary (i.e., the backup could wait until it has received enough log entries and then it could start applying them). However, in the case of distributed systems, the markers must be placed by the primary [5].

By applying the logs in groups, the backup can take advantage of batch processing optimizations. For example, multiple writes to the same page during the same group can be replaced by the last such write, thus reducing I/O traffic. Furthermore, the writes can be rearranged and applied in a different order than what is in the log, thus leading to better disk arm scheduling and reduced seek times. The order in which the updates are

applied may not correspond to any serializable schedule. However, as long as the state of the database after the application of an entire group is the same as the state that would have resulted if the log entries had been applied in sequence, consistency is preserved. Note that the application of updates in groups does not preclude continuous application; the latter can be viewed as a group application with group size equal to one.

The efficient application of updates may leave spare capacity for the processing of queries, and the database is consistent after the application of a group of updates. This leads us to the thesis of our processing scheme: read-only queries are always presented with the consistent version of the database that existed after the application of the updates of some group.

## 3. Classification of queries

In this section we attempt a classification of queries, based on aspects which affect the way in which the queries can be processed. In what follows, when we use the term queries, we mean read-only queries.

- A first classification is based on the currency requirements of queries. Queries are called *t-vintage* [6] if they want to see the version of the database which existed at a particular point in time $t$. Queries are called *t-bound* if they must read a version that reflects at least all transactions up to time $t$. An important special class of such queries results when $t$ is equal to the time that the queries are submitted, i.e., when queries want to see the effects of at least all transactions that ran before them. Queries are called *t-deadline* if they must see a version of the data that existed *before* time $t$. Queries are called $t_1 - t_2 - period$ if they want to view the data as of some time between $t_1$ and $t_2$. Finally, there are queries with no currency requirement.

- Queries are also characterized by their response time requirements. Queries with non stringent response time requirements can be delayed and run at a time that is most convenient for the system.
- Another important characteristic of queries is their size, or, in other words, the amount of data they access. Queries can be short (accessing only a few data items) or they may be long-lasting (batch queries). Batch queries typically do not have response time requirements and are usually t-bound, where $t$ is a relatively recent point in time. According to their access pattern, batch queries can be further divided into *ordered batch* and *random batch* [1]. Ordered batch queries need to process data in a particular order, while random batch queries can process their data in any order. For example, a query that needs to produce a monthly statement for each account in a bank can access the accounts in any order (random batch). The order in which an ordered batch query accesses the data may or may not coincide with the order in which the data are physically laid out on the storage media (disks). Queries that scan the data in physical layout order (scan queries) can usually be processed more efficiently, since they avoid long seek times.
- The final classification is based on the consistency requirements of a query. A query may not require any consistency at all, or it may require a consistent view of the data. In the latter case, it may be satisfied with weak serializability, or it may require strong serializability (this distinction applies to distributed systems [3], [6]). We assume that the database is always left in a globally consistent state after the application of the updates of some group.

We now examine what types of queries can be processed under our approach. Queries that do not require consistency can be processed at any time at the backup and

their execution does not interfere with other activity (e.g., groups can be installed concurrently, other queries may execute). For queries that require consistency, the currency requirement becomes significant. If a query is t-bound, one can present the query with the version of the database *after* the installation of the group into which time $t$ falls. This will ensure that all transactions that executed before time $t$ will be reflected in the copy that the query reads. Similarly, if a query is t-deadline, it can be presented with the version of the database *before* the installation of the group into which time $t$ falls.

In general, t-vintage and $t_1 - t_2$-period queries cannot be processed under our scheme. They can only be processed if the end of a group coincides with time $t$ (for t-vintage) or falls within $t_1$ and $t_2$ (for $t_1 - t_2$-period). In some cases it is possible to satisfy this condition by forcing the premature termination of a group, but we do not discuss this issue further.

The response time requirement may sometimes determine whether a query can be processed by our scheme. For example, if the group application of updates makes the data unavailable to queries for a period longer than the delay a query can tolerate, then the query cannot be processed under that scheme.

## 4. Algorithms for processing read-only queries

In the last section we showed that the requirements of many queries can be satisfied by presenting them with the version of the database after the complete installation of some group. In this section we show how this can be achieved. We focus our attention on queries which can complete within the duration of a single group. Queries that last very long and need a consistent view of the data are traditionally processed with multiversion schemes with more than two copies, which are beyond the scope of this paper. In our

analysis we assume that queries are ordered batch (but non-scan). Random batch and scan queries are easier to process and are discussed briefly in Section 7.

During a cycle, i.e., between the termination of two consecutive groups $n$ and $n+1$, a backup computer performs three tasks: it receives the logs for group $n+1$, it installs the updates for group $n$ and it processes the queries that need to view the database state after group $n$. The first task is *logically* independent of the other two and can be performed in parallel with them. Often it is also *physically* independent, since a separate disk may be used to hold the log (if the log is stored on disk; we discuss this issue later). However, the application of updates and the execution of queries may conflict with each other, since they may need to access the database in conflicting modes.

The query workload that is executed during a particular cycle can be divided into two classes: a portion of the query workload is available at the beginning of the cycle; we call such queries *prescheduled*. During the cycle, there may be incoming queries, which we call *incidental*. The system can decide to either execute an incidental query immediately (during the current cycle) or to defer it until the next cycle. The decision can be based on various parameters, e.g., the current load on the system, the response time requirement of the query etc. If the query is deferred for the next cycle, it becomes a prescheduled query for that cycle.

In what follows, we assume for simplicity that the logs contain the after images of entire pages. Such logs can be applied directly to the database. However, it is possible to have logs that need processing; for example, the log may contain logical actions or it may only contain the portion of a page that has been changed. In this case, the logs may need to be processed and the before images of pages may need to be read, thus adding to

the read traffic of the data.

We now examine various possible mechanisms to perform the updates and answer the queries during a cycle. We present four access conflict resolution techniques: separation of updates and queries, pointer conflict detection, blocking map conflict detection, and map conflict detection with on demand installation of updates.

The first approach is to separate the two tasks in time: the updates are performed first and then the queries are run (see Figure 1). While the updates are being applied, read access to the data is inhibited. While the queries are being executed, the data is left unchanged. The utilization of the disk bandwidth can be very high under this scheme. The application of all updates at the same time provides the disk scheduler with long I/O queues, which have been shown to increase throughput [10], [11]. If the logs do not require processing, the CPU may be underutilized while the updates are being installed,

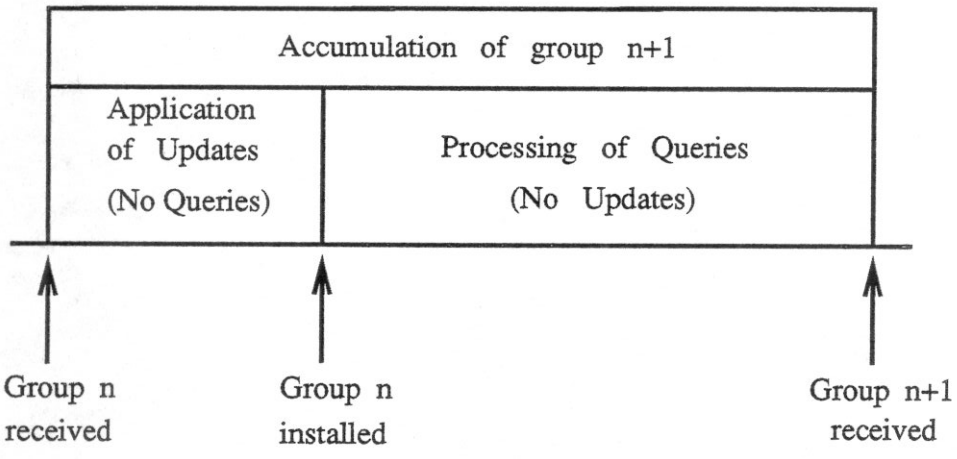


Figure 1. Separating Updates from Queries

since no queries are run during that time. Furthermore, since queries must wait for the installation of the updates, their response time may increase. The main advantage of this scheme is its simplicity: no synchronization overhead is incurred.

Another approach is to try to perform both tasks simultaneously. If a query must see a data item after group $n$, and that data item was modified by group $n$, then the query should be prevented from seeing the object until its new value has been installed. This implies that there must be a way of detecting access conflicts. One way is to apply the updates in the physical layout order of the data on the storage medium and keep track (using a pointer) of how far the update process has reached. (This is similar to many two-color algorithms found in the literature [8], [9].) A query can only read data that precede the current position of the pointer. Pseudo-code for this method is given in Figure 2. This option allows easy detection of conflicts, but it may deny access to data

---

```
Update process:
        pointer = address_of(first_data_item);
        while (there are more updates) {
                q = next_update; /* in physical layout order */
                write(q);
                temp = pointer;
                pointer = address_of(q);
                release all queries blocked between temp and pointer;
        }

Read request for page p by a query:
        if (p <= pointer)
                read(p);
        else
                block on p;
```

Figure 2. Pointer Detection of Conflicts

---

unnecessarily if the data is not modified by the current group but simply lies beyond the current position of the pointer. Also, the logs must be applied in physical layout order, which may be different from the order in which they were received.

Another way to detect conflicts is to keep a map of the pages modified by the group being installed. The map can be constructed as the logs are received and examined (and before any queries that need the new state of the data are processed). Note that the map may actually be implemented using some other data structure that provides the same functionality, possibly with better performance. Under this scheme, before a query can access a page, it must first check the map to see if an update is pending for that page. If an update is pending, then the query blocks until the new version of the page is installed by the update process. When the update process installs a new page, it resets the corresponding entry in the map. Pseudo-code for this technique is given in Figure 3.

```
Update process:
        while (there are more updates) {
                q = next_update; /* in any order */
                write(q);
                reset(map[q]);
                release all queries blocked on q;
        }

Read request for page p by a query:
        if (is_set(map[p]))
                block on p;
        else
                read(p);
```

Figure 3. Map Detection of Conflicts (blocking)

```
Update process:
        while (there are more updates) {
                q = next_update; /* in any order */
                write(q);
                reset(map[q]);
                release all queries blocked on q;
        }

Read request for page p by a query:
        if (is_set(map[p])) {
                install([p]);
                reset(map[p]);
                release all queries blocked on p;
        }
        read(p);
```

Figure 4. Map Detection (on demand update installation)

Yet another possibility is to detect conflicts using the map, but to resolve them on a demand basis: when a query needs to access a page for which an update is pending, the update is fetched from the log and installed prematurely (before the time at which it would normally be installed). Then the query is allowed to proceed. Pseudo-code for this case appears in Figure 4. This option improves response time, but it requires that the modified page be retrieved from the log out of sequence, which in turn implies that random access to the log is necessary.

In the pointer and map methods, the updates are installed concurrently with the query execution and the database contains a mixture of two versions. Since the updates are installed gradually, in the beginning of the cycle the database will be closer to version $n-1$ than to version $n$. This observation leads to an optimization we call dual copy access: an incidental query that arrives at the beginning of the cycle could be allowed to attempt to view version $n-1$. In this way, the query will not have to wait for updates to

be installed, but it will read older data. If the query comes across data that has already been updated by group $n$, the attempt fails; the query can be aborted and retried with version $n$.

A very interesting optimization is possible when all updates for a group can fit in the buffer cache (in main memory). While the log is being received, the new images of pages are stored in the database cache. When the group is to be installed, rather than write the pages on disk, one need only update the data structure that keeps track of the valid pages in the cache to reflect the presence of these pages and to invalidate the entries for the pages that have been modified. The modified pages can be written to disk at a convenient point (e.g., when the disk arm services a read request at the same or an adjacent cylinder). This will hide the cost of many writes under the cost of the reads, thus achieving significant performance gains.

## 5. Performance Evaluation

In this section we discuss the suitability and relative performance of the query processing schemes for various query workloads. We are not trying to predict the precise performance of systems under these algorithms. Rather, we attempt to provide more insight and understand the basic tradeoffs. We focus our attention on query throughput, not on response time.

We assume that the duration of a group is given. The factors that affect this choice are discussed in the next section. Without loss of generality we take this duration to be 1 unit of time. We adopt a very simple system model and consider only two computing resources, CPU and I/O. We restrict our attention to I/O operations on the disks holding the database; we do not consider the log disks.

A query performs $r$ steps, each involving some amount of CPU processing and some amount of I/O. The CPU has the capacity to process $A$ steps in a group period (unit of time), if no updates are present. Thus, the CPU can process a single step in time $\frac{1}{A}$. The disk takes time $\frac{\lambda}{A}$ to satisfy the I/O requirements of a step. The factor $\lambda$ characterizes the query workload, since it determines the relative utilization of I/O and CPU by a query. When a query initiates a step that cannot be processed immediately (because the data it wants to access is not available), the query blocks until the data it needs becomes available. A blocked query cannot initiate any other steps until the one it is blocked on is released (ordered batch). Furthermore, we assume that all queries are available at the beginning of the cycle (prescheduled queries). In what follows, we examine how many queries the various processing schemes can run in one group period.

The update process is assumed to utilize only I/O and have negligible CPU requirements. We also assume that the updates can be installed most efficiently if no read traffic interferes with them. Since updates are known in advance, they can be written in the physical layout order of the data to reduce seek times. We assume that all of the updates can be installed in time $u$ (without read interference). A random read request that is

| | |
|---|---|
| $u$: | time to install updates |
| $\lambda$: | relative utilization of resources |
| $p$: | penalty factor |
| $A$: | maximum # of steps per period |
| $r$: | steps per query |

Table I. System parameters

processed in the middle of the sequence of updates incurs an overhead, since it causes the heads to move to a different place to serve the read request and disrupts the efficient installation of updates. We consider this overhead part of the read request and assume that the I/O of a step executed during the update period takes time $\frac{\lambda p}{A}$ rather than $\frac{\lambda}{A}$, where $p$ is a penalty factor, $p > 1$. The system parameters are summarized in Table I.

First, we consider the case $\lambda > 1$, i.e., I/O bound queries. For this type of query workload, the separation algorithm is the most appropriate. Since the I/O capacity is the limiting factor, we would like to have as much I/O capacity available for queries as possible. This, in turn, implies that the updates must consume as little I/O capacity as possible, which according to our assumption can be achieved if the updates are installed separately from queries.

In the rest of the section we consider the case $\lambda < 1$, i.e., when the queries are CPU bound. If the separation algorithm is used for query processing, resource utilization would be as in Figure 5, where shaded areas correspond to utilized resources. From time 0 to time $u$, while updates are being applied, the CPU is idle, because no queries are executed. From time $u$ to time 1 CPU-bound queries are run and the CPU is saturated. During this period, only a fraction $\lambda$ of the I/O capacity is utilized, so that the I/O area is not entirely shaded in Figure 5. The number of query steps executed in this case is $A(1-u)$.

Since CPU cycles during the update period are wasted and CPU is at a premium, the pointer and map algorithms try to exploit this CPU idle period to run more queries. We start by examining map detection under a best case scenario: queries never block, i.e., they access either data that did not change during the group or data for which the new version has already been installed. This will give us an upper bound on the improvement

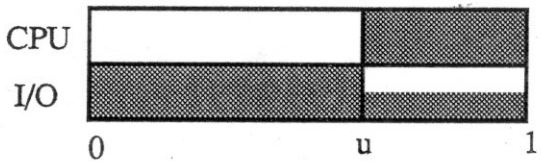


Figure 5. CPU Bound Queries

we can expect in the number of queries we can run within a group period.

Suppose that we move $t$ seconds of query CPU processing into the idle period. We also have to move into this period the corresponding amount of I/O, $\lambda t$. According to our model, the interference with the updates will multiply the cost of this I/O by the penalty factor $p$, so that the new cost of the I/O will be $\lambda pt$, and the update period will grow by that amount as well. The net effect of this arrangement will be a gain if the CPU time we moved into the update period (i.e., the time by which the query period shrank) is longer than the time by which the update period grew. This happens if $t > \lambda pt$, or $\lambda p < 1$.

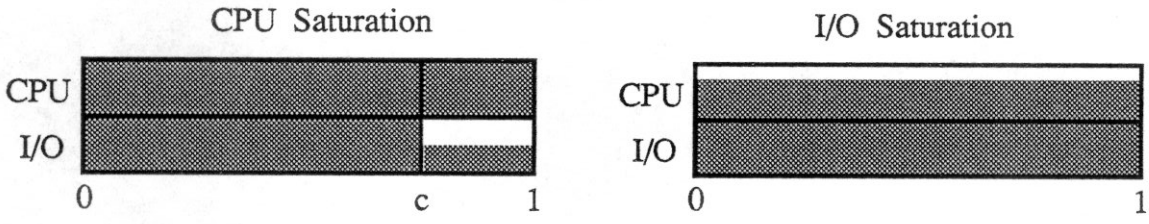


Figure 6. Saturation of CPU (left) and I/O (right)

If $\lambda p < 1$, moving query processing into the update period leaves some spare time in the query period, which can be used to run more queries. We can keep doing this to fit as many queries as possible into a group period. However, since the CPU utilization grows faster than the disk utilization (during the update period), at some point the CPU utilization will catch up with the disk utilization. This is shown in Figure 6 (left), where the catchup point (end of expanded update period) is denoted by $c$. We compute this point: $c = u + \lambda pc \Rightarrow c = \frac{u}{1-\lambda p}$. In this case the CPU utilization during the group is 100%, so we can run $A$ steps. The relative increase gained (with respect to the separation algorithm) in the number of steps executed during the group cycle is

$$G = \frac{A - A(1-u)}{A(1-u)} = \frac{u}{1-u}$$

However, depending on the values of the parameters, we may run out of disk capacity (i.e., the update period reaches the end of the group) before the CPU utilization catches up with the disk utilization (Figure 6, right). In other words, the catchup point may lie beyond the end of the group. This situation occurs if $\frac{u}{1-\lambda p} > 1$. In this case, only $qA$ steps get executed, where $q$ is the CPU utilization. To compute $q$ we note that $qA$ steps generate $qA\frac{\lambda}{A}p = q\lambda p$ of I/O activity. In addition, $u$ I/O time is taken by the updates and $u + q\lambda p = 1$ (I/O saturates). Thus, $q = \frac{1-u}{\lambda p}$. The relative throughput increase with respect to the separation case is

$$G = \frac{qA - A(1-u)}{A(1-u)} = \frac{1-\lambda p}{\lambda p}.$$

Figure 7 shows the relative increase $G$ for $u = 0.4$ (left) and $u = 0.7$ (right) as a function of $\lambda$. The curves shown correspond to $p = 2$ (solid line), $p = 3$ (dashed line) and $p = 5$

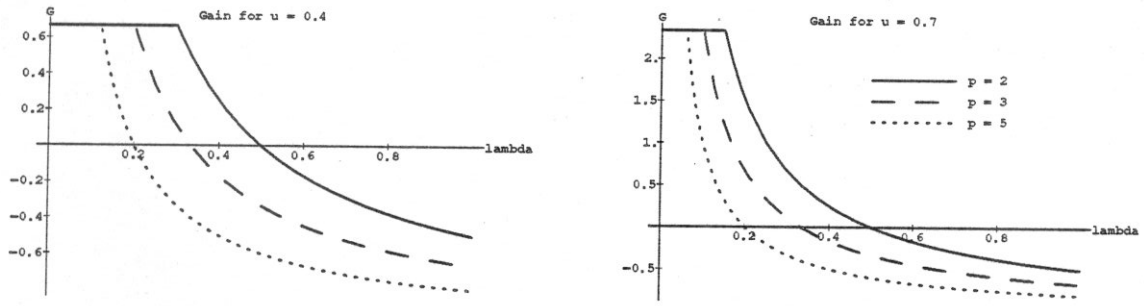


Figure 7. Relative Throughput Gain (map vs. separation)

(dotted line). The gain can be significant. For example, for $u = 0.7$, the maximum gain is over 200%. The gains get bigger as $u$ grows, since more CPU cycles are wasted in the separation method and exploited in the map method. The gains get smaller as $p$ grows, since each step needs more I/O time in the update period and I/O saturates with a smaller number of steps. Note that when $\lambda p = 1$, $G = 0$, i.e., the map and the separation method perform the same. If $\lambda p > 1$, the map algorithm is counterproductive. In the latter case, the I/O is always saturated and $G = \frac{1 - \lambda p}{\lambda p} < 0$, i.e., performance degrades. The slope of the curves shows the sensitivity of the results to the value of $\lambda$. The steeper the slope, the more sensitive the gain is to small changes of $\lambda$.

We now study the case of CPU bound queries when the pointer method is used for conflict detection. In the pointer method queries may have to block because the data they want to access lie beyond the position of the pointer. Thus, the CPU may have to be underutilized due to lack of workload. As we have mentioned, the maximum number of steps we could perform within a group period is $A$. Using the pointer method we can perform a fraction $v$, $v \leq 1$ of the maximum. In other words, we can execute $vA$ steps, which

correspond to $\frac{vA}{r}$ queries, since each query issues $r$ steps. Our goal is to calculate $v$.

Let $f(t)$ be the fraction of the data that has been scanned by the pointer, i.e., the fraction of the data that is immediately accessible to queries at time $t$. We define the backlog $B(t)$ to be the number of steps that must be performed after time $t$, because they will be issued by queries that are currently blocked. For example, if a query must issue 4 steps and at some moment it is blocked on its second step, it contributes 3 steps to the backlog at that moment. If we assume uniform access to the data, a step issued by a query succeeds with probability $f(t)$ and blocks with probability $1-f(t)$. Thus, when the pointer has scanned a fraction $f$, the probability that a query contributes $k$ steps to the backlog is the probability that the first $r-k$ steps of the query can be satisfied with data that precede the pointer, while the next (i.e., $r-k+1$) step requests data that lie beyond the pointer. We obtain the expected backlog by multiplying the average number of backlogged steps per query by the number of queries (i.e., $\frac{vA}{r}$):

$$B(f)=\frac{vA}{r}\sum_{k=1}^{k=r} kf^{r-k}(1-f)$$

In what follows, we always approximate the actual value of the backlog with its expected value. Figure 8 shows the (expected) backlog as a function of $f$ for $vA=1000$ steps and $r=4$ requests per query. In the beginning, since a very small portion of the data is accessible, most of the queries block and the backlog is almost equal to $vA$. The derivative $\frac{-dB}{dt}$ gives the rate at which steps are released by the advance of the pointer. In other words, this is the rate at which the CPU is presented with work to do. As long as this rate is lower than the capacity of the CPU (i.e., $A$), the CPU will be able to consume the released amount of work immediately and will be underutilized. At some crossing

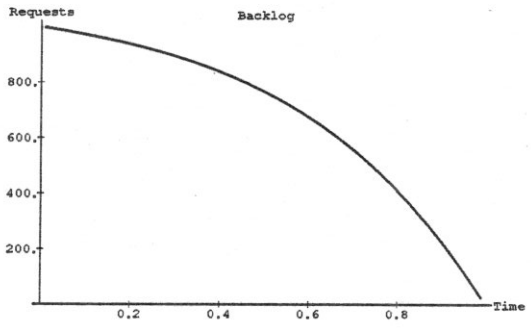


Figure 8. Backlog

point $t_c$, with a fraction $f_c$ scanned, the release rate will become equal to the capacity of the CPU, and from then on it will exceed it. After this point, the CPU utilization will be 100%. In order to compute this point, we must solve the equation $\frac{-dB}{dt} = A$, or $\frac{-dB}{df}\frac{df}{dt} = A$. We can replace the factor $\frac{df}{dt}$ by noticing that at the crossing point (and thereafter) the query processing activity saturates the CPU and therefore takes up a fraction $\lambda p$ of the I/O capacity. We know that originally (in the separation method) the updates were installed at a rate $\frac{1}{u}$. Now the updates will be installed by the remaining I/O capacity, i.e., at a rate $\frac{1-\lambda p}{u}$. If we assume that the updates are scattered uniformly on the data, the rate of update installation is also the rate at which the pointer is scanning the data. Thus, we can take $\frac{df}{dt} = \frac{1-\lambda p}{u}$ (at the crossing point), so that the equation $\frac{-dB}{df}\frac{df}{dt} = A$ becomes (the $A$'s cancel out):

$$\frac{v}{r}\left[\sum_{k=1}^{k=r} k f_c^{r-k} - \sum_{k=1}^{k=r-1} k(r-k) f_c^{r-k-1}(1-f_c)\right]\frac{1-\lambda p}{u} = 1 \qquad (1)$$

Equation 1 involves unknowns $f_c$ and $v$, so we need another equation. Time $t_c$ is the time it takes to install the fraction $f_c$ of updates plus the time it takes to satisfy the I/O requests for the query steps that completed before $t_c$. Since the installation of all updates requires time $u$, the installation of fraction $f_c$ requires $uf_c$. Before the crossing time, $vA - B(f_c)$ steps completed. These steps require $\frac{\lambda p(vA - B(f_c))}{A}$ I/O time. Thus,

$$t_c = f_c u + \frac{\lambda p(vA - B(f_c))}{A} \qquad (2)$$

After $t_c$ the CPU is fully utilized processing the remaining $B(f_c)$ steps, which take time $1/A$ each, so that

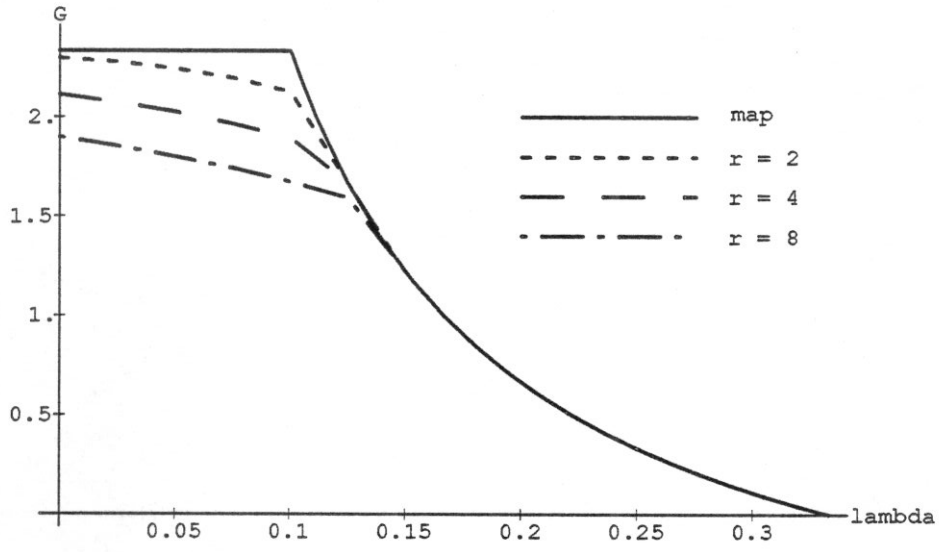


Figure 9. Gain in the pointer method

$$1 - t_c = \frac{B(f_c)}{A} \tag{3}$$

We combine Equations 2 and 3 to get:

$$\frac{(1-\lambda p)B(f_c)}{A} + f_c u + \lambda p v = 1 \tag{4}$$

Note that Equation 4 is independent of $A$, since the $A$ in the denominator cancels with the $A$ in the formula for $B(f_c)$. We can numerically solve simultaneous Equations 1 and 4 for $f_c$ and $v$. When we obtain the value of $v$, we can calculate the relative gain in throughput (with respect to the separation algorithm):

$$G = \frac{vA - A(1-u)}{A(1-u)} = \frac{v-(1-u)}{(1-u)} \tag{5}$$

For values of $\lambda$ such that $u + \lambda p \leq 1$, i.e., $\lambda \leq \frac{(1-u)}{p}$, the I/O never saturates and CPU utilization is limited only by the backlog effect. For $\lambda > \frac{(1-u)}{p}$, I/O saturates at some point $\lambda_s$. For values of $\lambda < \lambda_s$, our analysis holds. For $\lambda > \lambda_s$, our analysis does not hold, since the CPU is not saturated. It is difficult to characterize the point $\lambda_s$, so we proceeded by manually inspecting points beyond $\lambda = \frac{(1-u)}{p}$. If the I/O was not saturated, then Equation 5 was used. If I/O was saturated, the gain was the same as in the case of I/O saturation under the map method.

Figure 9 shows the relative increase in throughput (with respect to the separation method) as a function of $\lambda$, for $u = 0.7$, $p = 3$ and for $r = 2$ requests per query (dotted), $r = 4$ (dashed) and $r = 8$ (dot-dashed). Note that for the same value of $\lambda$ the gain decreases as $r$ increases, since the backlog effect becomes stronger. For comparison purposes, we also show the curve for the map detection (solid line). Remember that this is the best case curve for the map method (no blocking). The actual map method will block

on some accesses, but it will do so less frequently than the pointer method, since it will only block on modified data, not data that just happens to be located beyond the pointer. Thus, the pointer method gives us a lower bound on the performance of the map method, and the real map method curve lies between the pointer method curve and the best case map method curve.

## 6. Size of Groups

In this section we examine the factors that determine the duration of groups. A major factor is the takeover time. If a disaster occurs at the primary, the backup must complete the installation of the pending group and start accepting and processing incoming transactions. Thus, the takeover time requirement puts an upper bound on the time it takes to install a group, and, consequently, on the size of the group. Note that if updates are installed on a demand basis, it is possible to start transaction processing without having completed the installation of the group. Performance may be degraded in the beginning, since transactions may have to wait for the application of updates, but the takeover will not be delayed.

The query parameters also affect the group size. The group must be long enough to allow for the completion in a single cycle of the biggest query to be run under this mechanism. On the other hand, this may hurt the response time of queries that have to wait for the next cycle.

Throughput generally improves as groups become longer, since the longer the group the closer processing comes to batch mode, so the more efficient it can be made. However, certain optimizations are only applicable when the group length is limited. For example, the optimization that stores the updates in the buffer cache can only be applied

if the group size is small enough to fit all of the updates in the cache.

## 7. Conclusions

We have shown that primary/backup decoupling allows efficient batch installation of updates and simplifies query processing. We have presented several query processing algorithms and have evaluated their performance. Our analysis indicates that for CPU bound queries interleaving update and query I/O operations may be highly beneficial, improving throughput by a very significant factor. For I/O bound queries, separating updates from queries in time provides the best performance. In either case, it is more efficient to run the queries at the backup rather then the primary.

The performance model used is very simple, but in our opinion, is precisely what is needed to understand the key tradeoffs. For instance, the parameter $p$ captures very succinctly the cost of interleaving update and query I/Os. In practice, this depends on the actual disk scheduling algorithm, the disk geometry, the load, the seek functions, and so on. Introducing all these parameters and factors would obscure the issues. However, at a later stage when a particular system and application are known, a detailed model (or actual experiments) would be required to measure values of $p$ and other parameters.

To conclude, we now discuss some additional issues related to the algorithms we have presented. We mentioned that it is efficient to apply the updates in physical layout order. To avoid sorting the log explicitly, one can break the incoming log stream into substreams, each covering a contiguous part of the disk and having expected size equal to the buffer cache. When updates are installed, each substream is brought entirely into memory in turn, and its updates are applied in physical layout order.

In the case of multiple disks, our discussion applies to each disk separately. Since it is reasonable to assume that update installation proceeds in parallel and at the same rate on all disks, the fraction $f$ of scanned data for the entire database is the same as the fraction $f$ for each individual disk.

When the backup is distributed across several computers, each computer can use our techniques to install the local updates and run the local queries efficiently. Distributed queries (which access data at more than one computer) require that the database be in a *globally* consistent state after the installation of a group. This is a thorny issue in distributed query processing [3]. As we mentioned in Section 2, the epoch algorithm [5] can be used to achieve this property. The overhead of the epoch algorithm may be another factor in determining the size of groups.

An interesting issue is related to where the received logs are saved at the backup. If they are stored on stable storage, then they can survive crashes of the backup computer; their receipt can be acknowledged immediately to the primary, which can discard them. If the logs are stored in memory (as in the optimization presented at the end of Section 4), the primary cannot discard them until the updates have actually been installed at the backup; otherwise, in case of a crash at the backup the logs would be irretrievable. Furthermore, if the received logs are stored on disk, the system can tolerate a disaster at the primary with a simultaneous crash at the backup. If the logs are stored in memory, this scenario would result to loss of the last group received (and not installed).

In the previous sections we addressed ordered batch queries, with an order different from the physical layout order (non-scan). This is the worst case scenario for queries. For random batch queries (see Section 3) the performance is better, since the queries can

issue a subsequent access request out of order when the data for their current request is unavailable. Thus, there is essentially no blocking and performance gains are as shown in Figure 7. Queries that scan the database in physical layout order can be processed very efficiently: as the disk head moves to install the updates in layout order, it can also access the pages that the scan queries want to read. In our terminology, the penalty factor $p$ is 1, since the scan queries do not disrupt the installation of updates.

## REFERENCES


[1] R. Bayer, "Consistency of Transactions and Random Batch," *ACM Transactions on Database Systems*, Vol. 11, No 4, December 1986, pp. 397-404.

[2] D. Burkes and K. Treiber, "Design Approaches for Real Time Recovery," Presentation at the *Third International Workshop on High Performance Transaction Systems,* Pacific Grove, CA, September 1989.

[3] A. Chan and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 2, February 1985, pp. 205-212.

[4] H. Garcia-Molina and C. A. Polyzois, "Issues in Disaster Recovery," *IEEE Compcon*, San Francisco, CA, February 1990.

[5] H. Garcia-Molina, C. A. Polyzois and R. Hagmann, "Two Epoch Algorithms for Disaster Recovery," *16th VLDB,* Brisbane, Australia, August 1990.

[6] H. Garcia-Molina and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems*, Vol. 7, No. 2, June 1982, pp. 209-234.

[7] C. Mohan, K. Treiber and R. Obermarck, "Algorithms for the Management of Remote Backup Databases for Disaster Recovery," *IBM Research Report,* IBM Almaden Research Center, June 1990.

[8] C. Pu, "On-the-Fly, Incremental, Consistent Reading of Entire Databases," *Algorithmica*, 1986, pp. 271-287.

[9] D. J. Rosenkrantz, "Dynamic Database Dumping," *Proceedings of the ACM SIGMOD Conference on the Management of Data*, 1978, pp. 3-8.

[10] M. Seltzer, P. Chen and J. Ousterhout, "Disk Scheduling Revisited," *Proceedings Winter 1990 USENIX*, 1990.

[11] C. Staelin and H. Garcia-Molina, "Clustering Active Disk Data to Improve Disk Performance," Tech. Rep. CS-TR-283-90, Department of Computer Science, Princeton University, Princeton, NJ 08540, September 1990.

[12] Tandem Computers, *Remote Duplicate Database Facility (RDF) System Management Manual*, March 1987.