Database Server Organization for Handling Mobile Clients[1]

B. R. Badrinath Shirish Phatak
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
e-mail:{badri@cs, phatak@paul}.rutgers.edu

Abstract

The use of mobile computers is gaining popularity. The number of users with laptops, notebooks is increasing and this trend is likely to continue in to the future where the number of mobile clients will far exceed the number of traditional "fixed" clients. Applications running on the mobile clients download information by periodically connecting to repositories of data stored in either databases or file systems. Such mobile clients constitute a new and different kind of work load and exhibit a different access pattern than seen in traditional client server systems. Though file systems have been modified to handle clients that can download information, disconnect, and later reintegrate, databases have not been redesigned to accommodate mobile clients. There is a need to support mobile clients in the context of client server databases

This paper is about organizing the database server to take into consideration the access patterns of mobile clients. We propose a concept of **hoard key** which captures these access patterns. Three different techniques for organizing data at the server based on the hoard key are presented. We argue that each technique is suited for a particular workload. The workload is a combination of requests from mobile clients and traditional clients. This reorganization also allows us to address issues of concurrency control, disconnection, replica control in mobile databases.

We present simulation results that show the performance of server reorganization using hoard keys. We also provide an elaborate discussion of issues resulting from this new reorganization in this new paradigm that includes mobile clients and traditional clients.

1 Introduction

The use of laptops, notebook computers and PDAs is increasing and likely to increase in the future with more and more applications residing on these mobile systems. Many applications, such as databases, would need the ability to download information from an information repository and operate on this information even when the client is "out of range" of the repository, i.e., **disconnected**. These repositories are being accessed via dial up lines, cellular modems, CDPD, wireless lans or docking stations. An excellent example of this scenario is the mobile workforce. Such users would need information either from files from their home directories on a server or customer records from a database. The type of access and the work load generated by such users is radically different from the traditional workloads seen in client server systems of today.

For example, consider a utility company serving customers in the Bay Area that has a large mobile workforce of repairmen. These repairmen, who have laptops, need access to the corporate databases in order to access customer records. Typically, a repairman servicing customers in the

[1]This research work was supported in part by DARPA under contract number DAAH04-95-1-0596, NSF grant numbers CCR 95-09620, IRIS 95-09816.

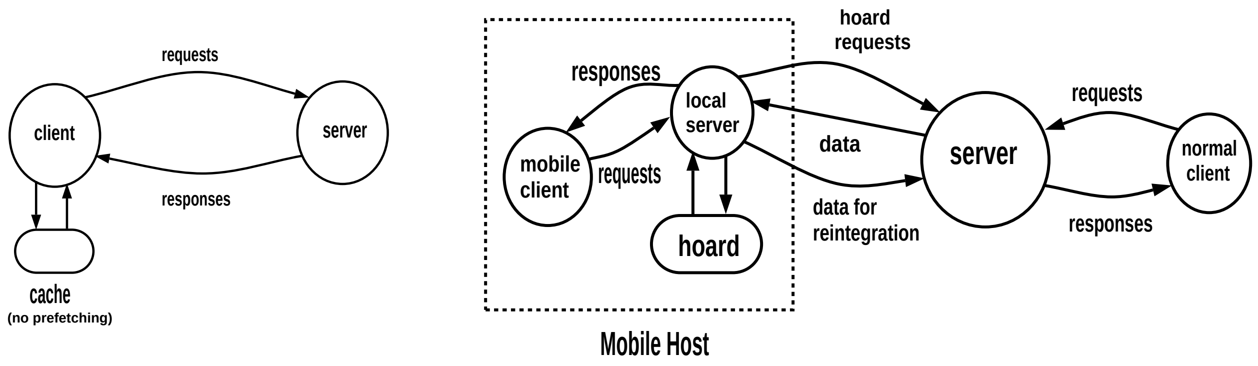

Figure 1: Client-Server and Hoard-Reintegrate Models

Palo Alto area would, only be interested in records of customers in Palo Alto. Based on the work done at the customer site, the repairman might want to update customer records while being disconnected (i.e., when the corporate database is not accessible). For example, the repairman can update the customer complaints database every time a complaint is resolved. For this purpose, the repairman needs the capability to download that part of the database that relates to the customers in Palo Alto onto her laptop, make changes while disconnected from the database server, and finally propagate the changes to the database server whenever the server becomes accessible via a network.

An important new feature in terms of work load in the above scenario is that the client hoards data and works locally on the data in a disconnected mode. In addition, the data hoarded by the client is likely to exhibit some locality of access. In the above example, a repairman servicing clients in Palo Alto will, typically, access and update only the data relating to customers in Palo Alto. This introduces a new type of client which hoards data based on some form of locality, operates on the data without the need of the database server, and later *reintegrates* the changes whenever the server can be accessed. These *hoard clients*, which use the hoard-reintegrate model also display some form of locality which might be different from other *general purpose clients*. We focus on the aspects of the server that need to be redesigned to facilitate hoarding and reintegration.

The model we consider here is a modified client server computing model (see Figure 1). As in the traditional client/server model (see Figure 1) each client has disk space and computing power that is an order of magnitude smaller than the server. A centralized database sits on the server and services requests from the client. Certain clients are allowed to hoard data on their local disks. When the client is fully connected, i.e., there exists a high bandwidth physical link between the client and the server, queries are serviced by the server. When a client disconnects, a local server uses the hoard to service any requests. Whenever the client reconnects, this local server resynchronizes its hoard with the server by reintegrating any local updates.

1.1 File Systems versus Databases

The model given here has been successfully implemented for File Systems such as CODA [20]. The granularity of hoarding chosen in CODA is an entire file. The client is allowed to hoard a set of files based on a hoard profile, e.g., all files in her home directory. The client can then perform local updates on the files after disconnection. On reconnecting, all updated files are reintegrated. If a file submitted for reintegration to the server has a hoard timestamp (i.e., the time at which the hoard client hoarded the file) later than the modification timestamp of the file on the server, the reintegration succeeds. If so, the client copy of the file replaces the server copy. Since the granularity of hoarding and reintegration is entire files each file is individually guaranteed to be consistent.

The CODA model has been extended to other applications using *Application Specific Resolvers* [13]. To use CODA semantics with databases, all tuples of the relations need to be mapped onto files. There are two distinct possibilities. Either an entire relation can be mapped onto a single file or subsets of tuples of a given relation can be mapped onto separate files. In the first case, each mobile client would carry the entire relation on its local disk. This is obviously not desirable since a typical mobile client would not have the resources (disk space) to hoard entire relations. Even if it did, most of the hoard data may fall outside the locality of access for the mobile client. In the second case, transaction atomicity is compromised since the unit of reintegration is tuples and not transaction updates. Hence, a mechanism is required to handle "slightly" inconsistent data while efficiently dealing with a hoard *which is an order of magnitude smaller than the size of the relation.*

If the database designer can pinpoint some form of locality in the access/update pattern of the clients, this locality can be used to cluster the data to allow for fast hoarding and to manage concurrency control issues at a finer granularity than entire relations. This is especially true if the locality is based on some keys in the relation. We call such a key a **hoard key**. In this paper, we consider various approaches to organizing the database based on hoard keys. These approaches have been evaluated using a simulation model in Section 4.

1.2 Organization of the Paper

The paper is organized as follows: Section 2 discusses related work. Section 3 describes the new server organization. The simulation model and discussion of results is presented in Section 4. Finally Section 5 presents conclusions and discusses new issues resulting from our approach to server organization for handling mobile clients.

2 Related Work

As mentioned in the introduction, hoarding is a concept that has been successfully applied to file systems. The CODA file system uses hoarding to allow users to work on their files even while disconnected from the server. The files are cached locally when the client is connected. On disconnection the cache functions as a local hoard. CODA uses the fact that it is unlikely that two users would modify the same file in a UNIX based environment. This is especially true when the users are modifying files owned by themselves. The CODA approach can be extended to other applications by allowing the application programmers to specify application specific resolvers that allow programmers to handle reintegration on a per application basis.

Another approach is provided by BAYOU [6]. Here the the hoard is replaced by a local copy of the *entire* database. Each host accessing the database must maintain such a copy. All the hosts are assumed to be intermittently connected. BAYOU defines an update as a triple consisting of a conflict detection mechanism, the write call and conflict resolution handler. The user performs updates on her local copy by specifying such a triple. A write goes though only if the conflict detection mechanism fails to detect conflicts, otherwise the conflict resolution handler is invoked. Each update made by the user since the last time when the database was known to be globally consistent is recorded in a log. Whenever two BAYOU hosts connect to each other all transactions in the log are undone and the logs combined in timestamp order. The combined log is used to replay the updates on both hosts and to make the databases mutually consistent. BAYOU guarantees that in absence of any fresh updates the database will ultimately converge to a consistent state on all hosts as long as the hosts do not remain disconnected for ever. The responsibility of handling conflicts between updates is left to the user and the system does not provide any global concurrency control.

Another suggestion is an update anytime, anywhere model proposed in [8]. The database here is a collection of replicated objects with primary copies at certain sites known as object masters. The model distinguishes between mobile nodes that remain disconnected most of the time and base nodes which are alway connected. The objects are replicated on a set of nodes using a two tier replication scheme. One tier exists on the mobile whereas the other tier is maintained on the base nodes. Each mobile node maintains a master copy of the database which is the most recent validated copy received from the object master and a tentative copy on which updates are performed. Each read-write transaction on the mobile node is considered tentative. All tentative transactions are sent to the base node on reconnection along with a set of acceptance criteria. If the results of the base transaction and the tentative transaction satisfy the acceptance criteria, the base transaction writes the master copy and commits.

None of the above work deals specifically with server organization of data to accommodate mobile clients. However the idea of reorganizing databases for specific applications is not new. For example, in *Multidimensional Online Analytical Processing* or *MOLAP*[5], a flat relational database is reorganized into a multidimensional read-only store to support analytical queries involving operators such as min, max and sum. Each dimension holds the result of applying some operators to the database. Most analytical queries can be executed efficiently on the multidimensional store merely by slicing out one or more of the dimensions. This paper deals with analogous reorganization of the server database for mobile clients.

3 Data Organization on the Server

With the introduction of mobile clients, database servers can expect two distinct workloads. First, the workload generated by hoard requests from mobile clients. Second, the workload generated by general purpose queries from traditional clients. Mobile clients hoard data from the server, act on the data locally (in disconnected mode) and later reintegrate updated data with the server. On the other hand, traditional clients submit queries to the server; the server executes the query and returns the results. For such workloads, typically the primary index decides the database layout on disk (i.e. the mapping from tuples to pages on disk) to offer better access time.

In this paper, we propose a new database organization to handle workloads generated by mobile clients. Under our new organization, the database designer can specify a set of *Hoard keys* along with the Primary and Secondary keys for each relation. Hoard keys are supposed to capture typical access patterns for mobile clients.

Each hoard key partitions the relation into a disjoint set of logical horizontal fragments[4, 23]. These fragments constitute the hoard granularity, i.e., hoard clients can hoard and reintegrate within the scope of these fragments. There are two distinct mechanisms to provide access to these fragments. In the first mechanism, the database can maintain an index data structure for each logical fragment associated with a hoard key. In this case the hoard key is a *logical hoard key.* A logical hoard key does not affect the physical organization of the database. The logical fragments are identified by special index data structures that instantiate the mapping between the tuples and the logical fragments.

In the second mechanism, the database is physically organized to mirror the logical fragments. In this case, the hoard key is a *physical hoard key.* Physical hoard keys control the physical layout of the database. The tuples in the database are clustered by the logical fragments associated with the hoard key. If there is exactly one physical hoard key, then each cluster or physical fragment corresponds to a logical fragment associated with that hoard key. Physical hoard keys allow faster access to data which a given mobile client would be interested in hoarding. This is because the data which a mobile client is likely to access is physically contiguous on disk and thus can be accessed efficiently. However this organization may affect the performance of general purpose queries because the tuples are no longer clustered on the primary index. Furthermore, each physical fragment can be made autonomous, i.e., each fragment can have its own server, indexing strategy and concurrency control policies.

Having defined hoard keys, the next issue is to define the mapping of tuples to the logical fragments. This mapping is specified by defining a set of qualifying relations or qualifiers. Each qualifying relation defines a logical fragment. Qualifiers are similar to the **WHERE** clause of a **SELECT * FROM R WHERE condition** statement where **R** is the relation and **condition** is a predicate on the hoard key. Thus, the qualifying relations specify "aggregations" of key values. If no qualifiers are explicitly specified, each value in the domain of the hoard key is associated with its fragment. The qualifiers provide a mechanism for fine tuning locality of access/update as well as controlling the size of the fragments. A tuple is in a fragment iff the value of the hoard key for that tuple satisfies the qualifying relation for that fragment. Each fragment specifies a logical cluster of tuples. However, the physical organization of the database depends on the number of physical hoard keys in the database. In addition to providing a mechanism for logical clustering of tuples, fragments are the granularity at which concurrency control and reintegration are managed

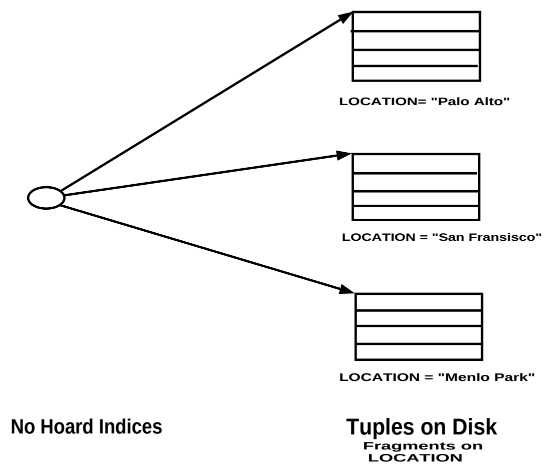

Figure 2: Single physical Hoard Key

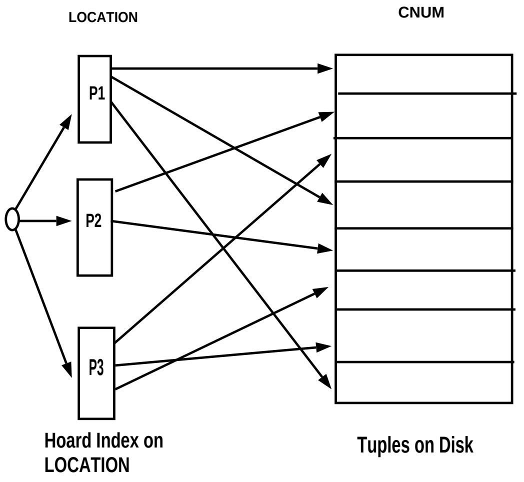

Figure 3: Single logical Hoard Key

by the server.

The following schema and qualifiers for a single hoard key illustrate logical fragments:

CUSTOMER(CNUM, LOCATION, TYPE)

Primary Key is **CNUM**,

Secondary Key is **TYPE**,

Hoard Key is **LOCATION**.

The qualifiers for **LOCATION** can be as follows:

- Q_1: **LOCATION** = "Palo Alto"
- Q_2: **LOCATION** = "San Francisco"
- Q_3: **LOCATION** = "Menlo Park"

Here tuples for all customers in Palo Alto would be placed in fragment Q_1, San Francisco in Q_2 and Menlo Park in Q_3. If **LOCATION** is a physical hoard key, then the tuples of the database are clustered into three fragments; one corresponding to each logical fragment (see Figure 2). If the hoard key is logical, the tuples are ordered by the primary key, and indices can be built for the three fragments (see Figure 3). By specifying a fragment id, the client can request to hoard these fragments. Since all tuples belonging to the fragment are clustered together, the fragment can be retrieved efficiently.

In general, there can be multiple hoard keys some of which are logical and some others physical. With hoarding, there are essentially three different possible database organizations corresponding to three possible cases: 1) where all hoard keys are logical, 2) where all hoard keys are physical and 3) where some hoard keys are physical and some other hoard keys are logical. The physical organization of the database depends on the number of physical hoard keys. For example, consider the following schema modeled from [4] with two hoard keys:

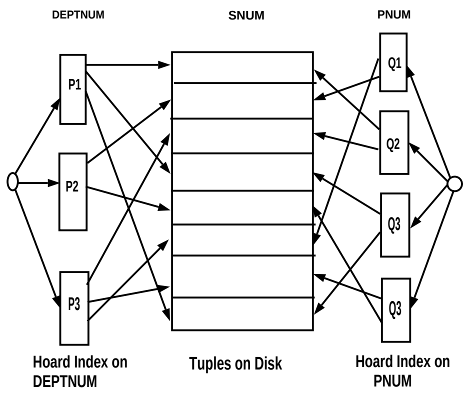

Figure 4: Both hoard keys are logical

SUPPLY(SNUM,PNUM,DEPTNUM,QUAN)
Secondary Key is **SNUM**,
Hoard Keys are **DEPTNUM** and **PNUM**

The domain of **DEPTNUM** is $[0, \ldots, 30]$ and that of **PNUM** is $[0, \ldots, 4000]$. Let us assume that the qualifiers for **DEPTNUM** are:

- P_1: **DEPTNUM** < 10
- P_2: **DEPTNUM** $>= 10 \bigwedge$ **DEPTNUM** < 20
- P_3: **DEPTNUM** $>= 20$

Let us assume that the qualifiers for **PNUM** are:

- Q_1: **PNUM** < 1000
- Q_2: **PNUM** $>= 1000 \bigwedge$ **PNUM** < 2000
- Q_3: **PNUM** $>= 2000 \bigwedge$ **PNUM** < 3000
- Q_4: **PNUM** $>= 3000$

These seven qualifiers define seven logical fragments. Though the logical fragments defined by any one hoard key are disjoint, logical fragments defined by different hoard keys might overlap. For example a tuple having **DEPTNUM** $= 9$ and **PNUM** $= 2000$ would be in both P_1 and Q_3. Such logical fragments interact depending on whether all hoard keys are physical, logical or a combination of the two.

There are three distinct cases to consider:

- **Case 1**: All the hoard keys are logical hoard keys.

  In this case, the tuples in the database are organized by the primary index. For each hoard key, each fragment defined on that hoard key is realized by constructing an separate index for tuples in the fragment.

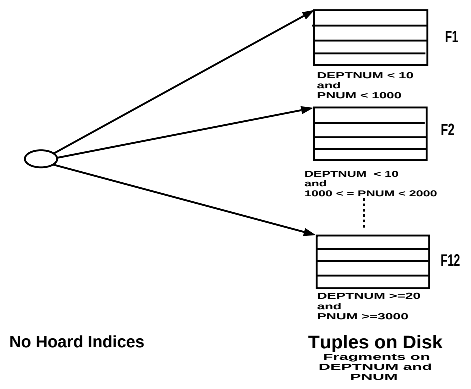
F1
DEPTNUM < 10 and PNUM < 1000
F2
DEPTNUM < 10 and 1000 < = PNUM < 2000
F12
DEPTNUM >=20 and PNUM >=3000
No Hoard Indices
Tuples on Disk
Fragments on DEPTNUM and PNUM

Figure 5: Both hoard keys are physical

For example, assume that **DEPTNUM** and **PNUM** are both logical hoard keys. This case is illustrated in Figure 4. Here the tuples in the database with consecutive values of **SNUM** are stored in a physically contiguous fashion. For each of the qualifiers P_i's, Q_i's listed above, a index is created on the corresponding hoard key indexing all tuples satisfying the qualifier. For example, for P_1 an index is created on **DEPTNUM** indexing all tuples satisfying the condition **DEPTNUM** < 10. There are a total of 7 logical fragments, 3 for **DEPTNUM** and 4 for **PNUM** .

This scheme can be used when the workload primarily consists of traditional clients with the possibility of a few hoard clients. Since the physical organization of the database is unchanged, the access time seen by the traditional clients remains unaffected. The indexes defined on the logical fragments provide faster access to hoarding clients.

- **Case 2**: All the hoard keys are physical hoard keys.

  The database on disk is physically organized into fragments each of which contains tuples satisfying conjuncts of the qualifying relations[2]. The number of physical fragments is the product of the number of logical fragments for each hoard key.

  For example, consider both **DEPTNUM** and **PNUM** as physical hoard keys (see Figure 5). Here the database is physically fragmented into 12 fragments corresponding to the conjunctions of each of the P_i's with one of the Q_j's. Here fragment F_1 corresponds to **PNUM** < 1000 and **DEPTNUM** < 10, fragment F_2 corresponds to $1000 \leq$ **PNUM** < 2000 and **DEPTNUM** < 10 and so on. Note that in this case, each physical fragment represents an update locality.

  This scheme can be used where there are large number of clients modifying data based on both **PNUM** and **DEPTNUM** . Also, the database designer can use the qualifiers to reduce the

[2]Since all the qualifiers in the set of qualifiers for a given hoard key denote mutually exclusive conditions, the only meaningful conjuncts have *all* qualifying relations from different sets. For example a conjunct involving any two of the P_i's would be a contradiction. Thus for the example in this section the only conjuncts that could specify nonempty sets of tuples are the ones involving exactly one P_i and one Q_j. Another way of saying this is that only the logical fragments associated with different hoard keys can overlap.

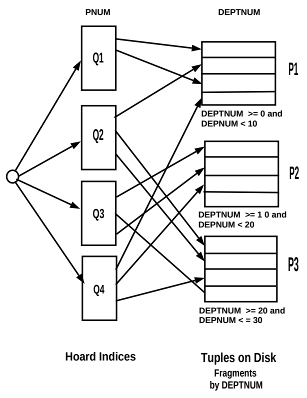

Figure 6: One of the Hoard Keys is physical

size of each physical fragment to an extent where *entire fragments can be hoarded by the mobile host.* Thus the database designer can completely eliminate most indexing on hoard keys. This is because every hoard request by the client can be satisfied simply by downloading the entire physical fragment corresponding to the clients update locality[3]. Furthermore, the server can emulate CODA behavior by accepting the entire fragment on reintegration from a client that has updated *some* tuples in the fragment or by rejecting all the updates.

Because the tuples with consecutive primary key values are no longer contiguous a dense index must be built for the primary key; i.e., an index record would appear for every primary key value in the relation. Any query accessing tuples by primary key would suffer since consecutive key values can access different physical fragments.

- **Case 3**: Some of the hoard keys are physical and some of the hoard keys are logical.

  Here, the organization of data is a hybrid of cases 1 and 2. All the physical hoard keys together define the organization of data on disk as in case 2. The logical fragments for all the logical hoard keys are indexed as in case 1.

  For example, consider **DEPTNUM** as a physical hoard key and **PNUM** as a logical hoard key as in Figure 6. The database is physically fragmented for each P_i. In this case there are three fragments consisting of tuples with **DEPTNUM** with values in the range $[0,\ldots,9]$, $[11,\ldots,19]$ and $[20,\ldots,30]$ respectively. Furthermore an index is created for each of the Q_i's.

  This scheme can be used when most hoard clients access data based on the physical hoard keys (**DEPTNUM**) but there are a few clients that access data by the logical hoard keys (**PNUM**). The indices on logical hoard keys in each fragment can be used to recreate any logical fragment. These fragments may then be cached by the server to facilitate hoarding.

The tuples within each physical fragment can be organized by one of **PNUM**, **DEPTNUM** or **SNUM** . Since **PNUM** is the second hoard key (and is presumably "less" important in the sense that less clients are likely to request data based on **PNUM**) it is unlikely that ordering tuples

[3]indexing is still required to support general purpose queries

by **PNUM** would give any performance improvement. If the number of values that the physical hoard key can have within the fragment is large, then the clustering of the data based on physical hoard key would be a sensible choice. However if the number of values is small then the data can be organized using some other key, such **SNUM** .

Both **DEPTNUM** and **PNUM** might have duplicate key values since these are not primary keys. Thus any indexing strategy for these keys would have to deal with duplicates.

3.1 Discussion

When we have physical hoard keys as in cases 2 and 3, the database is composed of many disjoint physical fragments. Each fragment in effect can be treated as a separate autonomous database. A central database server could be responsible for multiplexing client queries to each of the autonomous database servers. This central server would also plan the execution of general purpose queries and send the plans to each of the component servers. The advantage of this approach is that each of the autonomous servers could use their individual access strategies and policies for indexing and organization of tuples on the disk. In effect the database could be a distributed database and each fragment could be placed at different sites. It would be the database designers responsibility to decide the indexing schemes for each fragment.

For systems mostly serving general purpose clients, it makes sense to have few physical fragments. Hence, the database designer needs to make all hoard keys logical or to make the fragments as large as possible. As the ratio of hoarders to general purpose clients increases, it is beneficial to have smaller fragments and more physical hoard keys. Thus for systems mostly serving hoard clients, all hoard keys should be physical hoard keys and the fragment should be small. In particular, if most hoarders are likely to hoard data on a certain hoard key, then that that hoard key must be made physical while keeping all other keys as logical hoard keys. Thus the concept of a hoard key provides the database designer with a rich set of options to configure the system. Exactly how to determine the hoard keys and the qualifiers for a given workload is an interesting research issue.

The client can hoard data by using hoard queries. Such queries need only refer to some fragment identifier rather than the relation. The server (if there are many autonomous databases, then the server responsible for the fragment) dynamically builds indices for checked out data for each client. This allows the database to efficiently track checked out data and also to perform locking operations. The database can use the indices to efficiently locate tuples submitted for reintegration by clients and also to decide whether or not the tuple is in the clients update locality.

4 The Simulation Model

The primary purpose of the simulation was to determine the impact of hoard keys on performance for each of the three cases described in Section 3. Performance was measured on the basis of the number of disk accesses required to fetch a tuple from the disk, including index accesses. These statistics were collected for general purpose queries as well as hoard queries. General purpose

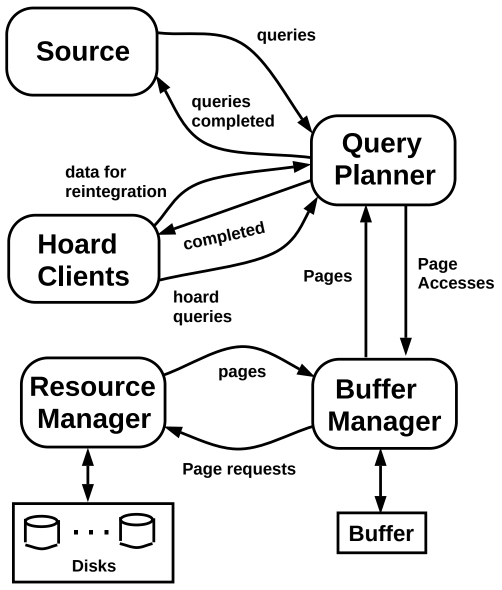

Figure 7: The Simulation Model

queries are modeled as range queries on the entire database while hoard queries are modeled as requests for tuples that belong to a fragment. The simulation model is illustrated in Figure 7. The model consists of the following components:

- *Source*: Source of all general purpose queries.
- *Hoard Clients*: Source of all hoard queries and the disconnection manager.
- *Query Planner*: Plans page accesses for queries, including index accesses. The query planner assumes multiprogrammed single CPU operation.
- *Buffer Manager*: Manages the buffer cache. All page requests are first directed to the buffer manager.
- *Resource Manager*: Manages the *Disks*. All page requests ultimately come to the resource manager when they can not be satisfied from the buffer cache.

4.1 Simulation Assumptions and Design

Since the primary goal was to measure disk access performance for the three organizations on a centralized database, the simulation did not take into account CPU times. The database is modeled as a single relation mapped onto a collection of *DBSize* pages with *TuplesPerPage* tuples in each page. There are two hoard keys which define two sets of logical fragments. The sizes of these sets is given by two parameters *NumFragments1* and *NumFragment2*. As described in section 3, the number of physical fragments on disk depends on the organization of the database. Each of these fragments are striped across *NumDisks* disks. The sizes of the hoard queries are assumed to be uniformly distributed between $0.5*AvgHoardQuerySize$ and $1.5*AvgHoardQuerySize$. The sizes of general purpose queries is assumed to vary uniformly between 1 and $2*AvgGenQuerySize-1$.

The source "thinks" for an random exponentially distributed time with mean *ThinkTime* before generating the next query. Independent sources generate general purpose and hoard queries. The mean think time is divided between the two sources by the parameter *FractionGeneral*. The general purpose think times are further scaled to reflect the ratio of the average size of hoard query versus the average size of a general purpose query. A query generated by the general purpose query source is "hot" with a probability *GenQueryHotProb*. A *hot* general purpose query is defined as a query that accesses consecutive tuples ordered by the primary index. A "cold" query accesses tuples at random. In each case the tuples are picked at random from different physical fragments. The concept of hot and cold access is defined in [2]. We assume that within a physical fragment the tuples are ordered by primary key and that there is uniform distribution of tuples across physical fragments.

The hoard query source generates a sequence of hoard queries. The source also tracks clients that need to reintegrate some fraction of their data on reconnection. Each hoard query generated by the hoard query source is given an "update locality" which is the fragment identifier of a physical fragment. In the simulation we consider general purpose queries made by hoard clients to be *cold* queries. A query from a hoard client is *hot* with a probability of *HoardQueryHotProb*. Here a hot query is defined as a query accessing tuples only within the "update locality" of the fragment. *FractionFirst* fraction of hot queries hoard on the first hoard key, the remaining queries are on the second hoard key. A cold hoard query is treated in the same fashion as a cold general purpose query; all the requests are uniformly distributed over the relation, and not just a single fragment. After a hot query is serviced it is placed in a disconnect queue for a random, exponentially distributed disconnect time with mean *HoardDisconnectTime* and then submitted for reintegration. Note that this parameter only makes sense for *hot* hoard queries.

The queries generated by both the sources are then sent to the query planner which estimates the overhead of accessing the index and generates a stream of page requests. The overall overhead is calculated to be random number between 1 and the logarithm of the size of the fragment for accessing the first tuple (assuming a tree based index), plus an overhead for accessing the index for the remaining pages. The distribution of the page requests is controlled by the number of fragments and the server organization (which one of the three possible cases discussed in Section 3). For general purpose queries the tuples are assumed to be uniformly distributed over fragments. Every tuple is mapped onto a fragment by performing a simple hashing operation on the tuple number. For hot queries submitted by mobile clients, the requests lie within the same logical fragment. In the case where only the first hoard key is physical, the logical fragments for the second hoard key are scattered *across* physical fragments. Thus requests for a hoard query accessing tuples by the second hoard key are distributed across the disk as in the case of general purpose queries.

The query manager passes the request to a buffer manager which manages a buffer pool. A cache hit leads to immediate completion of the request, otherwise the buffer manager submits the

Parameter	Description	Value
DBSize	Number of Pages on Disk	100000
NumDisks	Number of Disks	10
DegreeMulti	Degree of Multiprogramming	10
TuplesPerPage	Number of Tuples in a Page	100
NumFragments1	Number of Logical Fragments associated with the First Hoard Key	$1 - 10$
NumFragments2	Number of Logical Fragments associated with the Second Hoard Key	$2 - 10$
AvgHoardQuerySize	Average Number of Tuples accessed by a Hoard Query	1000
AvgGenQuerySize	Average Number of Tuples accessed by a General Purpose Query	$100 - 900$ *(mostly* 100*)*
FractionGeneral	Ratio of the Average Number of Tuples accessed by a General Purpose Query to Average of Total Number of Tuples accessed by both Hoard and General Purpose Queries	$0.2 - 0.9$
GenHotQueryProb	Fraction of General Purpose Queries that are Read-Only Queries	0.8
HoardHotQueryProb	Fraction of Queries from Hoard Clients that access data only within the clients locality	0.9
FractionFirst	Fraction of Hoard Queries that hoard on the First Hoard Key	0.8
ThinkTime	"Combined" average think time for the query sources	5 *minutes*
HoardDisconnectTime	Average amount of time for which a hoarder disconnects after hoarding	15 *minutes*

Table 1: Parameters used in the simulation model and their values

request for a disk access.

4.2 Simulation Results

We conducted simulation experiments to determine the performance of various server organization schemes. Each experiment consisted of 10 runs for a simulated period of 4 hours for each of the cases described in Section 3. The database size was fixed at 4MB with 40 bytes/tuple and 100 tuples/page. The indices had 100 nodes/page with each node containing 5 key values. The average size of the hoard query was fixed at 1000 tuples and that of a general purpose query to 100 tuples. The mean think time was 10 secs. 20% of all hot hoard queries hoarded data based on the second hoard key. The set of simulations was run for each of the organizations presented in the previous section. In each run, the average number of disk accesses required to fetch a tuple has been plotted against the number of logical fragments associated with the first hoard key.

In the case where both the keys were logical (Figure 8) the number of logical fragments had little

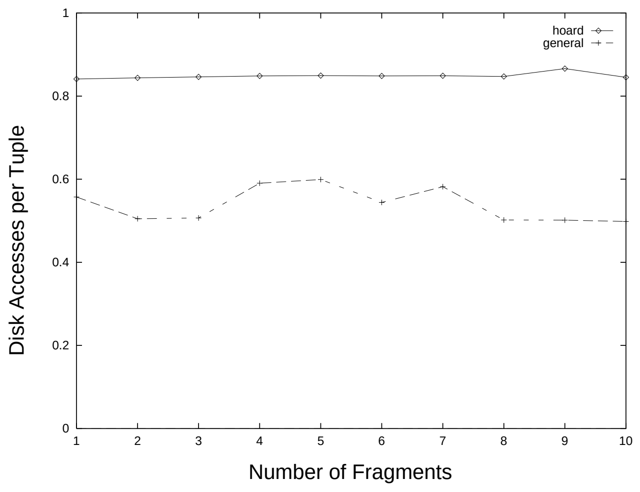

Figure 8: Both keys logical

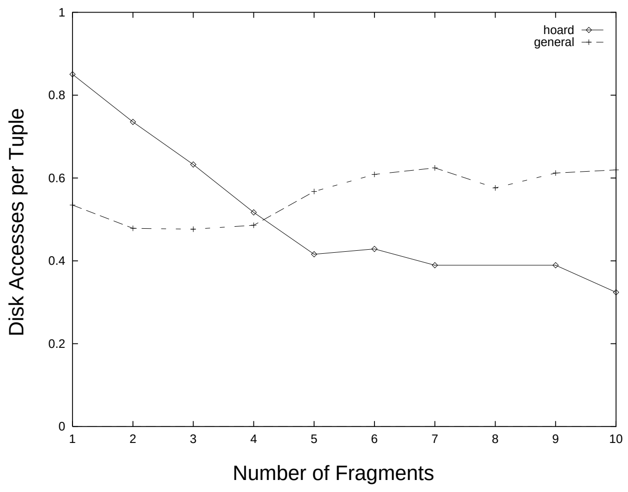

Figure 9: One key physical, One key logical

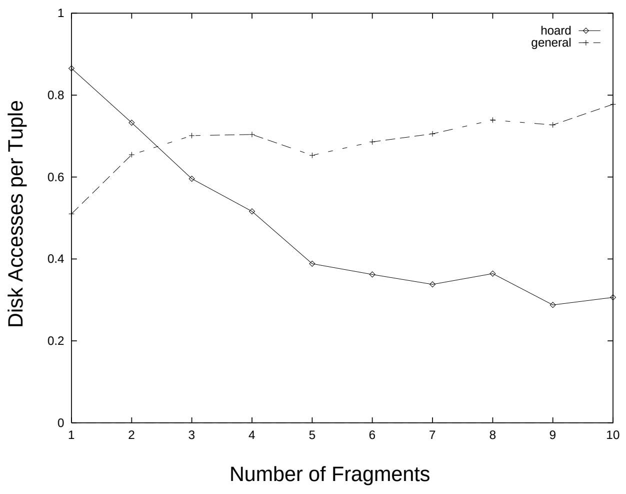

Figure 10: Both keys physical

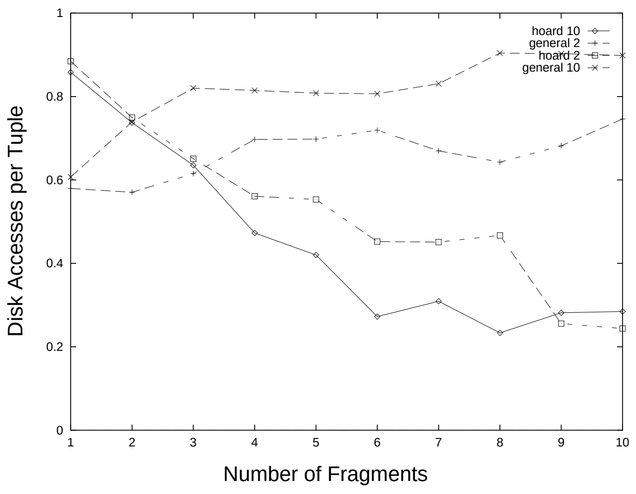

Figure 11: Both keys physical

impact on the performance of both general purpose queries and hoard queries. Since the database is not physically fragmented, the general purpose queries remain unaffected by the number of logical fragments. The insensitivity of the hoard queries to the number of fragments is due to the fact that logical fragmentation only reduces the size of the index per logical fragment. Since the index access overhead is very low compared to the number of disk accesses required to access a tuple, the size of the index has negligible effect.

When first hoard key is a physical hoard key (Figure 9) the situation changes dramatically. The number of disk accesses required to access a tuple drops from about 0.85 to about 0.35 for hoard queries as the number of fragments varies from 1 to 10. At the same time the number of disk accesses required for general purpose queries goes up from 0.5 to 0.7. At about 3 fragments the performance of both types of queries is about the same which would be ideal for a system designed for roughly equal numbers of mobile and general purpose clients.

If both keys are physical hoard keys (Figure 10), the above behavior is even more marked. The graph shows a plateau at 7 logical fragments which corresponds to 35 physical fragments (since there are 5 logical fragments on the second hoard key). Beyond this point not much is gained by increasing the number of fragments. In the case where both keys are physical hoard keys, the number of physical fragments on disk is a product of the number of logical fragments defined by both the hoard keys. The above experiments were conducted with the number of logical fragments of the second hoard key set to 5.

We also conducted an experiment with varying number of logical fragments corresponding to the second hoard key. Figure 11 shows the performance when the number of logical fragments on the second key was 2 (labeled *hoard 2* and *general 2*) and 10 (labeled *hoard 10* and *general 10*) respectively. The performance of the general purpose queries is affected by the number of logical fragments on the second hoard key. The performance with 10 logical fragments on the second hoard key is worse than with 2 logical fragments on the second hoard key. However, the performance of the hoard queries for 10 logical fragments (on the second hoard key) is better than with 2 logical fragments (on the second hoard key) only when the number of logical fragments on the first hoard key is at least 3 fragments. The number of fragments on the first hoard key is a dominant factor in determining the performance of the hoard queries. This is because only 20% of the hoard queries refer to the second hoard key.

As can be seen from the graphs, increasing the number of physical fragments significantly improves the hoard query performance, while slightly penalizing general purpose queries. For a given cost model and any given mix of general purpose queries and hoard queries, the performance of disk access time can be used as a guideline for organizing the database (e.g., deciding on the number of fragments).

5 Conclusions and Future Work

The results of this paper indicate that the server database can be reorganized to provide faster disk access for mobile clients. The number of fragments can be carefully controlled to suit the needs of a given application. This new organization where “hints” in the form of hoard keys and qualifiers are provided to the server raises the following research issues:

- **Performance Tuning**

  There is a tradeoff between the performance of general purpose clients and hoard clients. This tradeoff needs to be carefully evaluated to produce optimal performance. There is also the problem of specifying the qualifying relations. These relations could be a priori specified by the database designer or could be dynamically determined based on the access patterns. In the latter case, the index data structures and the physical layout of the database might change dynamically. Thus, efficient schemes for tracing and evaluating database activity and dynamically reorganizing the database would be needed (see [12]). Furthermore, hoard profiles could be generated for all the users of the database. How exactly to reorganize the database for a given set of hoard profiles is an open research problem.

- **Concurrency Control**

  There are also the problems of managing concurrency control. The issue of concurrency control is especially crucial since we allow local updates on the mobile clients. Since these clients are often disconnected, i.e., there is no information flow possible from the client to the server, the client is unable to synchronize any operations with the server. Thus, either the synchronization must be performed a priori while hoarding or during reintegration. A very common model that uses the former is the checkout/checkin model[11] that locks all the tuples hoarded by a client. The latter approach can be implemented by performing some form of time-stamping and comparing timestamps on reintegration. Here an additional complication arises from the fact that reintegration must be done at the granularity of a client transaction, i.e., client should be able to reintegrate any updated data in order defined by the transactions that performed these updates.

  With fragmentation, the database administrator would have the ability to specify what sort of concurrency control to use in each logical fragment. Each client can negotiate a contract with the server for some guarantees with respect to hoarded data. For example the server might allow tuples from a low contention fragment to be checked out using the checkin/checkout model, while allowing weaker forms of hoarding on other fragments. The decision whether to allow checkout or not can be made on a per fragment as well as on a per client basis. For authorized clients the server would allow checkout, but other clients might not be provided strong guarantees.

- **Reintegration**

  Efficient reintegration would be a major concern, especially when large amounts of data need to be reintegrated. In our model it might be possible for both the server and the client to make some decisions to aid reintegration. For example the server might require that the client resynchronize its hoard within a particular time frame. If a client does not reintegrate its modifications in this time frame, the server would reserve the right to reject transactions from the client.

Even though high bandwidth cheap links are not always available, low bandwidth ubiquitous links, albeit more expensive, slower links such as a cellular are available. Such links can be used to perform limited synchronization between the client and server, such as locking, tuple invalidations to flag a tuple as modified and trickle reintegration to reintegrate low volume updates[17]. Such solutions are interesting even without disconnection since increasing number of users are connecting to information servers via low bandwidth dial up lines. Since the connection is the bottleneck, it would make sense to perform extensive prefetching so that the database can be operated on locally on the client while the connection is used for low volume reintegration.

If the client has sufficient resources to hoard entire fragments then the database can use a CODA like approach to reintegrate data[4]. Depending on whether the servers copy of the fragment is more recent or not, either all the updates from the fragment would be reintegrated or none would be reintegrated.

- **Query Miss**

  Another problem with disconnected operation is the problem of determining whether or not the query was completely satisfied from the local hoard. If a query performed on the hoard produces more tuples when performed on the entire database a *query miss* is said to have occurred. Detection of query misses is especially important where the logical fragment sizes are much larger than the size of the hoard. Hence, a general mechanism for detecting query misses is required. Hoarding of metadata can be used in detecting query misses (for example range queries). Another way of preventing query misses is to require that all queries is disconnected mode refer only on the local hoard and not to the entire fragment.

- **Tracking Hoarders**

  From the servers point of view there is the issue of tracking who has checked out what. The server must detect and resolve conflicts between client updating within the same locality. To do so the server needs to keep track of the tuples checked out by each client for hoarding. One way to do this is to build dynamic indices for each client that hoards data. However this solution does not scale well with the number of clients. Thus some mechanism is required to "cheaply" track hoarded tuples. Such tracking mechanisms might also be useful for efficient reintegration.

- **Managing Fragments**

  Introducing logical fragments in the relation also brings about the issue of managing these fragments. The database designer needs to decide on indexing strategies to realize logical

[4]The server might be able to use this approach even when the client hoards only a part of the fragment. This can be done by ensuring that the part of the fragment not hoarded by the client has not been modified.

fragments corresponding to logical hoard keys. If some hoard keys are physical, the database designer also has an option to use different fragment servers to manage the physical fragments. In particular, each physical fragment might even be on geographically distinct sites. The interaction of logical hoard keys with physical hoard keys needs to be studied for each of these options.

References

[1] R. Alonso and H. F. Korth, Database System Issues in Nomadic Computing, Proceedings of the ACM SIGMOD, Jun. 1993, pages 388–392.

[2] K. P. Brown, M. J. Carey and M. Livny, Goal-Oriented Buffer Management Revisited, *Proceedings of the ACM SIGMOD*, Jun. 1996, pages 353–364.

[3] M. J. Carey, M. J. Franklin, M. Livny, E. J. Shekita, Data Caching Tradeoffs in Client-Server DBMS Architectures, *Proceedings of the ACM SIGMOD*, May 1991, pages 357–366.

[4] S. Ceri and G. Pelagatti, *Distributed Databases—Principles and Systems*, McGraw-Hill, 1984.

[5] E. F. Codd, E. S. Codd and C. T. Salley, Beyond Decision Support, *Computerworld* 27:30, July 1993, pages 87–89.

[6] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer and B. Welch, The BAYOU Architecture: Support for Data Sharing Among Mobile Users, *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications*, Dec. 1994, pages 2–7.

[7] M. J. Franklin, B. T. Jonsson and D. Kossmann, Performance Tradeoffs for Client-Server Query Processing, *Proceedings of the ACM SIGMOD*, Jun. 1996, pages 149–160.

[8] J. Gray, P. Helland, P. E. O'Neil and D. Shasha, The Dangers of Replication and a Solution, *Proceedings of ACM SIGMOD*, Jun. 1996, pages 173-182.

[9] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, 1993.

[10] T. Imieliński and B. R. Badrinath, Mobile Wireless Computing: Challenges in Data Management *Communications of the ACM*, 37(10), 1994, pages 18–28.

[11] R. Katz and S. Weiss, Design Transaction Management, *Proceedings of the 21st Design Automation Conference*, 1984, pages 692–693.

[12] G. Kuenning, G. J. Popek and P. Reiher, An Analysis of Trace Data for Predictive File Caching in Mobile Computing, *Proceedings of the USENIX Summer Conference*, 1994, pages 291–303.

[13] P. Kumar and M. Satyanarayanan, Supporting Application-Specific Resolution in an Optimistically Replicated File System, *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Oct. 1993, pages 66–70.

[14] N. Krishnakumar and R. Jain, Mobile Support for Sales and Inventory Applications, in *Mobile Computing*, T. Imieliński and H. F. Korth Ed.

[15] M. L. McAuliffe, M. J. Carey and M. H. Solomon, Towards Effective and Efficient Free Space Management, *Proceedings of the ACM SIGMOD*, Jun. 1996, pages 389–400.

[16] K. Mogi and M. Kitsuregawa, Hot Mirroring: A Method of Hiding Parity Update Penalty and Degradation during Rebuilds for RAID5, *Proceedings of ACM SIGMOD*, Jun. 1996, pages 183–194.

[17] L. B. Mummert, M. R. Ebling and M. Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access, *Proceedings of the 15th ACM Symposium on Operating System Principles* 29(5), Dec. 1995, pages 143–155.

[18] P. E. O'Neil, The Escrow Transactional Method, *ACM TODS* 11(4), Dec. 1986, pages 405–430.

[19] P. E. O'Neil, *Database—Principles, Programming, and Performance*, Morgan-Kaufmann, 1994.

[20] M. Satyanarayanan, CODA: A Highly Available File System for a Distributed Workstation Environment, *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, Sep. 1989, pages 447–459.

[21] A. Silberschatz, H. Korth and S. Sudarshan, Database System Concepts, McGraw-Hill, 1997.

[22] V. Srinivasan and M. Carey, Performance of B-Tree Concurrency Control Algorithms, *Proceedings of the ACM SIGMOD*, May 1991, pages 416–425.

[23] M. Tamer Özsu and P. Valduriez, Principles of Distributed Database Systems, Prentice Hall Inc., 1991.

[24] G. Walborn and P. Chrysanthis, Supporting Semantics-Based Transaction Processing in Mobile Database Systems, *Proceedings on the 14th Symposium on Reliable Database Systems*, Sep. 1995.

[25] G. Walborn and P. Chrysanthis, Transaction Processing in Mobile Computing Environment, *IEEE Workshop on Advances in Parallel and Distributed Systems*, Oct. 1993, pages 77–82.