# An Architecture for Mobile Databases[1]

B. R. Badrinath Shirish Hemant 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 and notebooks is increasing and this trend is likely to continue in the future where the number of mobile clients will far exceed the number of traditional fixed clients. Applications running on mobile clients download information by periodically connecting to repositories of data. Mobile clients constitute a new and different kind of workload and exhibit different access patterns than those seen in traditional client server systems. Though file systems have been modified to handle clients that are capable of downloading information, updating the information while disconnected, and later reintegrating the updates, databases have not been redesigned to accommodate mobile clients. Thus, there is a need to support mobile clients in the context of client server databases.

In this paper, we present a new architecture for database systems which takes mobile environments into consideration. This architecture allows us to address issues of concurrency control, disconnection, replica control in mobile databases. We also propose the concept of **hoard keys** which facilitates hoarding. We present simulation results that illustrate the performance of an example database system with both disconnected 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**. An excellent example of this scenario is a mobile workforce. In this scenario users would need to access and update 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 significantly 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. Based on the work done at the customer site, the repairman would want to update customer records while being disconnected (i.e., when the corporate database is not accessible). In this example, the repairman would update the customer complaints database every time a complaint is resolved. For this purpose, the repairman needs the capability to replicate 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.

An important new feature of this scenario is that the client *hoards* data and works locally on the data in a disconnected mode. We call such clients *hoard clients* as opposed to *traditional clients* that need 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.

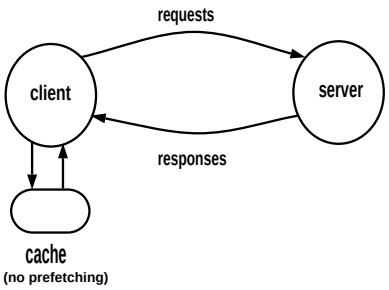


Figure 1: Client-Server Model

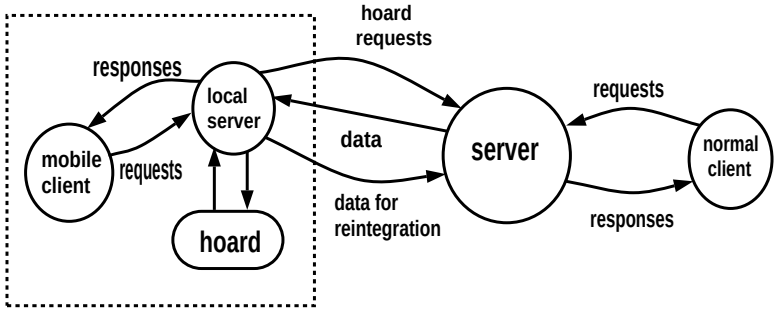


Figure 2: Hoard-Reintegrate Model

server to operate. The data hoarded by the hoard clients 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.

In this paper, we focus on those aspects of the database server that need to be redesigned to facilitate hoarding and reintegration. The model we consider here is a modified client-server computing model (Figure 2). As in the traditional client-server model (Figure 1) each client has limited resources as compared to the server (disk space and computing power). A centralized database server services requests from clients. Hoard clients are allowed to hoard data on their local disks creating local replicas. When a hoard 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 on the client uses the hoard to service requests. Whenever this client reconnects, local servers reconciles its replica with the servers copy by reintegrating any local updates. In this paper we use the terms hoard and local replica interchangeably.

## 1.1 File systems versus Databases

This modified client-server model has been used for file systems such as CODA [24], and has been extended to other applications using *Application Specific Resolvers* [17]. The granularity of hoarding in CODA is an entire file. The client is allowed to hoard a specified set of files, e.g., all files in a users home directory. The client can then disconnect and perform local updates on the files. On reconnecting, all updated files are reintegrated. If a file submitted for reintegration to the server has a hoard timestamp (the time at which the hoard client hoarded the file) later than the modification timestamp of the file on the server, the reintegration succeeds and 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.

To use CODA 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 not desirable since a typical mobile client would not have the resources (disk space) to hoard large relations. Moreover, most of the hoard data would fall outside the *locality of access* for the mobile client. In the second case, transaction atomicity is compromised since the unit of reintegration would be tuples or arbitrary groups of tuples and not transaction updates. Note that both

these problems occur with any disconnected system which manages replication on the granularity of files. Hence, a mechanism is required to handle "slightly" inconsistent data (all updates made while a client is disconnected lead to inconsistency) while efficiently dealing with hoards *that could be an order of magnitude smaller than the size of the relation.*

## 1.2 Replication versus Hoarding

The local hoard on hoard clients is in effect a local replica of a part of the database. Thus the hoard-reintegrate database is a special case of a replicated database. Replicated database systems have already been studied in some depth in literature [2, 6, 26]. Unfortunately, all these systems assume that there is a persistent connection between all the replica servers and any disconnection is due to failure. Thus, these solutions work poorly in a mobile disconnected environment.

We have considered the following options for hoarding:

1. *Database on a disconnected replicated file system*: These systems suffer from the same problems faced by a database on the CODA file system.
2. *Traditional Client-Server Architectures*: In these architectures the client relies on the server for all data processing. Thus, these architectures require a persistent connection, possibly limited in bandwidth, between the server and the client. This is unacceptable in an environment where disconnection is a norm.

   Even when the client uses an optimistic caching scheme—where updates can be deferred till commit, and the queries can operate on the local cache—the server must still service read requests from the client. Furthermore, the server is the only entity in the system capable of committing transactions. Each commit must be referred back to the server. Moreover, all the schemes in literature ([5, 12]) require significant amount of information flow between the server and the client. In a mobile scenario, such a flow is impossible. Thus, a simple caching scheme is not sufficient.
3. *Fully Replicated Database System*: A fully replicated database system requires that the entire database be replicated on the "clients" (which would actually be servers in this scenario). As mentioned before, typical mobile clients do not have the resources to deal with large relations, hence this solution is limited to small databases. Creating a new replica (hoarding) is a costly operation since the entire database (or at least a relation) must be replicated. These systems also suffer from all the problems of a traditional distributed database system.
4. *Traditional Distributed Database System*: In these systems the data can be replicated at the granularity of fragments. Since the fragments can be made small, this scheme can incorporate mobile clients. Creation of replicas (hoarding) is a relatively cheap operation. However, mobile clients are forced to replicate entire fragments. Traditional distributed systems also view network partitions as failures, and are typically pessimistic in their approach to overall consistency. This is unacceptable in a system where disconnection is a norm rather an exception.

   Even for traditional distributed database systems using optimistic consistency models, all servers replicating the data (or at least a majority of them) must be available for a transaction to commit. This is impossible in a disconnected mode of operation.

Thus, we need a database system that satisfies the following goals:

1. A full fledged relational database should be provided to mobile systems. Transactions should be supported and ACID properties should be provided to the extent possible. Information about any violation of these properties should be propagated back to the application.
2. Mobile clients should be not be forced to replicate any fixed amount of data; rather these clients should be allowed to replicate as much data as they see fit.
3. Fast hoarding should be supported. For example, a client wishing to replicate a small part of a relation should not have to wait for the entire relation to be shipped.
4. As far as possible, detection and resolution of update conflicts should be application/user transparent. Obviously, the database system would not have all the semantic information required to perform conflict resolution. However, conflict detection and some conflict resolution can be performed in context of the concurrency control model (see section 3.3)
5. Idiosyncrasies of mobile environments should be accounted for. For example, the mobile database system must account for disconnection and the fact that mobile clients are resource constrained.

We would also like to support "traditional" clients. These clients do not have any concept of replication of data, and do not understand or operate correctly under weakened ACID guarantees. Therefore we add the following design goal

- Traditional clients should not be penalized in a system shared by both hoard and traditional clients. In particular, though hoard clients may not be provided full ACID guarantees, traditional clients should be provided all the guarantees. Moreover, any reorganization of data should not significantly impact performance of traditional clients.

In this paper, we describe a system that achieves these goals. Our system does not provide all the ACID guarantees to transactions running on disconnected clients. In particular, our system can guarantee only local consistency (i.e. consistency on a single replica) to transactions running on the disconnected client and cannot guarantee durability. We assume that applications/users are aware of this constraint.

### 1.3 Organization of the Paper

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 describes our new architecture. Section 4 describes our model for fragmentation of relations. A simulation model and discussion of results is presented in Section 5. Finally, section 6 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. Examples of such systems are CODA and FICUS [16]. These systems allow clients to update replicas of files

while disconnected, and reconcile updates on reconnection. CODA uses a single server to reconcile updates, whereas FICUS uses a peer-to-peer mechanism.

In BAYOU [9], the hoard is replaced by a local copy of an *entire* data repository. Each server must maintain such a copy. These copies can then be accessed by other hosts connecting to such a server. All the servers are assumed to be intermittently connected. BAYOU defines an update as a triple consisting of a conflict detection mechanism, a write call, and a conflict resolution handler. Each component of the update triple must be defined by the application. A write goes though only if the conflict detection mechanism fails to detect conflicts, otherwise the conflict resolution handler is invoked. Each update made since the last time when the database was known to be globally consistent is recorded in a log. Whenever two BAYOU servers connect to each other, all transactions in the log are undone and the logs are 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 the absence of any fresh updates, the database will ultimately converge to a consistent state on all servers, as long as the servers do not remain disconnected for ever. The responsibility of handling conflicts between updates is left to the application and the system does not provide any global concurrency control. Furthermore, the system itself is not a full fledged database system, and can provide only very weak guarantees for applications ([27]).

In the update anytime, anywhere model proposed in [11]. 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, that are always 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 tentative transaction on the master copy 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*[7], 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.

Our approach differs from these approaches in that we provide a full fledged relational database model, but we do not require the client to hoard an entire relation or database. Furthermore, conflict detection is application transparent. The application/user is informed about the presence of conflicts and can resolve them. The system can also be specifically designed to provide automatic conflict resolution for certain applications. In a manner analogous to MOLAP, we create a tree like organization of the relations where nodes are fragments of a relation and the leaf nodes are the physical fragments on disk.

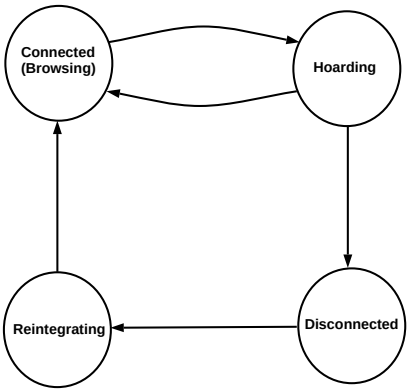


Figure 3: State Diagram

# 3 Database Architecture

## 3.1 Basic model

Our model consists of a centralized database with two types of clients: traditional clients and hoard clients. Hoard clients are capable of downloading (hoarding) data, operating on the hoarded data and then propagating (reintegrating) updates to the server. Since the clients can independently operate on the hoarded data, they are capable of operating even when there is no connection to the server. Traditional clients can not operate on the data on their own and rely on full connectivity to the server for their operation. The state diagram for a hoard client is given in Figure 3.

A hoard client hoards data from the centralized server in form of horizontal fragments of the relations[2]. Thus each client is not required to replicate entire relations, which solves scalability problems. The client may then disconnect from the server. Applications/users can continue to query and update the fragment on the client. While connected, the hoard client behaves like traditional clients and is allowed full access to the database. Thus, a mobile application can browse the database and decide on the data to be downloaded. Hoarding can be initiated explicitly (say through a begin_hoard primitive) or implicitly (data downloaded during browsing can be treated as the hoard).

Reintegration can be performed using any partition healing algorithm that can work with just two servers participating. In particular Davidson's partition healing algorithm [8] or CODA like approaches can be used. In each case the unit of reintegration can either be all updates from the hoard client or updates by individual transactions from a hoard client.

With the introduction of hoard clients, database servers can expect two distinct workloads. First, the workload generated by hoard requests from hoard clients. Second, the workload generated by traditional queries from traditional clients. The workload presented by hoard clients is bursty in nature due to their mode of operation. For each hoard client, the server first sees a burst of reads (hoarding) followed by a period of no activity or little activity (disconnection or weak connection), followed by a burst of validations and writes (reintegration)[3].

[2] note that as long as the primary key for the relation is hoarded for all practical purposes we can treat mixed fragments as horizontal fragments.

[3] Similar bursty behavior is also seen in traditional client server systems using optimistic cache validation techniques. However

The server is the point of synchronization in our architecture. All updates must finally be propagated to the server. The server also controls access privileges to various clients. Entire relations are stored on the server (as opposed to the hoard clients which can store parts of the relation). Physical organization of the relations on the server is in terms of physical fragments in order to aid fast reintegration and hoarding. Note that transactions from traditional clients only run on the server. From the server's point of view all hoarding and reintegration operations are performed at the granularity of fragments (though the clients may not necessarily hoard/reintegrate entire fragments). This reduces bookkeeping by not requiring the server to keep track of exactly what was hoarded by each client and improves the scalability of the system.

### 3.2 Organization of data on the server

As mentioned in the introduction we believe that typical hoard clients demonstrate locality of access. To capitalize on this fact the relations on the server are horizontally fragmented to optimize hoarding and reintegration performance. Our simulations (see section 5 and [3]) indicate that this can be done without significantly impacting time to access disk of traditional clients, even though such clients do not demonstrate any such locality of access. Note that performance of such clients is still affected by the resources consumed by hoarding operation. We also assume that clients know a priori what data they might need access to. For example in our repairman example, the repairmen know the geographic area that they are going to serve. It is possible to work around this assumption using hoard profiles[16]. Here the client need not explicitly specify which fragments to download. Instead, the client, say a sales manager on her way to a meeting in Singapore, can indirectly ask for data, as in, "download everything related to Singapore". The database system can then use the hoard profile for *Singapore* to determine exactly which fragments to download. Clients would also be able to determine what to hoard by browsing the relations in connected mode.

These physical fragments must be carefully designed to capture the locality of access of "typical" hoard clients. This needs analysis of the workload offered by such hoard clients. As our results in [3] indicate, a well organized relation can lead to an improvement of upto three time in raw i/o performance for hoard queries, without significantly affecting traditional i/o performance. We present our techniques for fragmenting relations using *hoard keys* in section 4. In general any well tuned mechanism for fragmentation can be utilized.

Physical fragments are the unit of hoarding, reintegration, concurrency control and access control in our model. Each hoard client is required to inform the server about the fragments it intends to utilize for hoarding data. A client does not need to hoard the entire fragment; however, from the server's point of view the client hoards entire fragments. It is the server's responsibility to resolve conflicts due to overlapping hoards and to provide some measure of conflict resolution. The server must also control access to each fragment and maintain a mapping between clients and the fragments they have hoarded.

in these systems the server sees a stream of reads followed by a burst of writes (at commit of *each transaction*). Moreover we expect the bursts to be far larger in a mobile database system than in traditional client-server systems, since the unit of hoarding can be as large as a fragment, and reintegration involves propagation of updates from a *number of transactions*. Note that on each client the workload would follow the traditional clients, however this workload would not be visible to the server.

### 3.3 Concurrency Control and Consistency Model

Since we allow disconnected clients to locally update the data sets, our architecture inherently has weak consistency. Furthermore, different clients might try to update the same datum in different ways. For example, consider the utility company database mentioned in the introduction. Consider two repairmen serving the same geographic area, say Palo Alto. Both repairmen might try to update the same customer's records in conflicting ways. For example if the customer registers the complaint, the first repairman to visit the customer might flag the complaint as pending if she can't resolve the complaint. However the second repairman might visit the same customer, resolve the complaint and mark the same complaint as resolved. Thus the global database state becomes inconsistent and the server has to deal with conflicting updates.

There are two approaches for dealing with consistency problems in the face of imminent disconnection. The first approach is pessimistic and requires explicit synchronization before disconnection to maintain consistency. An example of this approach is the checkin/checkout model [14] where the server simply locks all the data hoarded by the client. This approach eliminates conflicts at the cost requiring too much information in advance, and reducing availability of the system. The second approach is optimistic and allows conflicting updates on all disconnected hosts and resolves the conflicts on reconnection. This approach is difficult to implement and can potentially suffer due to high volume of conflicts. However it is attractive since it provides high availability of the system, which is precisely what we require. Here we describe an optimistic approach to the consistency problem.

Conflict detetection is implicitly peformed by doing serializability testing of the reintegrating transactions. This can be done using Davidson's algorithm, or by using CODA like semantics (e.g. reject a transaction, if any of the data items that it "touched" were updated after the client hoarded). Fragment level information can be used to provide fast verification. For example, tuple level validation (where the granularity of detecting conflicts is a single tuple) need only be done for a reintegrating client if the fragment was updated after the client hoarded data. In fact, in low contention fragments, we could dispense with tuple validation altogether, and reject all transactions from a client if the fragment was updated since it was last hoarded by the client.

Note that we clearly separate the local concurrency control from the global consistency model. Local concurrency control on the server and the clients can be any standard concurrency control model including pessimistic 2-phase locking. In fact, the server and the clients need not even be using the same concurrency control model—rather the clients could be using models better suited to their end users and applications. Nevertheless the overall global consistency model needs to be optimistic to ensure overall availability of the database.

To reintegrate data from hoard clients, the server needs to provide conflict detection and resolution mechanisms. We take the approach that conflict resolution is primarily (but not entirely) the concern of the application. In particular, the database does roll back any conflicting transactions, but does not automatically replay or otherwise compensate for the such transactions' updates. Information about rolled back transactions must be propagated back to the application issuing the transaction, if the application so wishes. On the other hand, conflict detection must be built into the server. In our architecture, the global consistency model implicitly provides conflict detection.

In our global consistency model, a copy of each data item (which in our case is a tuple) is designated the

master copy, and all committed transactions that update the data item must commit on this copy. Conflicts are also detected by checking whether or not a transaction can be serialized on this copy. Note that this must be done for all the data items that are updated by the transaction. If the transaction can be serialized on the master copies of all the data items updated by it, it is conflict free and can be committed. Otherwise it is considered conflicting and must rolled back along with any transactions that read dirty data from this transaction. Such transactions can be replayed later at the discretion of the application or the user.

In our model the granularity for management of consistency issues is a physical fragment. Accordingly the data items are collectively mastered as a data-set in terms of an entire physical fragment. The restriction is that each data item must belong to exactly one such physical fragment, i.e. each data item must be mastered exactly once. This implies that the physical fragments for a relation must not overlap. Any commit on the master physical fragment is considered "final" or "globally committed" and can not be rolled back. Commits on any other copy of physical fragment are considered "locally committed" and can be rolled back if any conflicts arise. Thus, in our architecture any transaction operating on a non-master fragment is not guaranteed durability. This is similar to the isolation-only transaction model proposed in [19], except that atomicity is automatically guaranteed by the local server on the client. This is because the local server guarantees "local" ACID properties on the local replica of the fragment.

If a master physical fragment is on a client rather than a server, then committing transactions on the server becomes complicated. In particular, if a transaction's updates span physical fragments that are mastered on different machines, then the updates must be propagated and serialized on all machines for the transaction to globally commit. Furthermore, care must also be taken while creating the master physical fragments on various machines, since only one machine can have the master physical fragment. In spite of these problems, support for migrating master replicas is required to support certain applications and transaction models, such as the checkin/checkout model [14].

Transactions from traditional clients cannot be subjected to rollback since traditional clients will always assume durability of committed updates. Thus the server must also ensure that a traditional transaction never commits on a non master copy of the data. There are two ways of ensuring this: 1) Do not allow traditional transactions to enter local commit state or 2) whenever a traditional transaction attempts to access non master data, it should be blocked until the master copy becomes available. Unfortunately, both the approaches reduce effective availability of the database at the server. This is a compelling reason for restricting hoard clients from hoarding master copies of fragments for long periods of time.

# 4 Creating and defining fragments: Hoard Keys

Since fragments play such a crucial role in our architecture, we also need to provide efficient ways to define and create fragments. Furthermore, we would like to keep the solution as flexible as possible, allowing the the database administrator to refine/add fragments as and when needed with minimum effort[4]. Furthermore, we would like to ensure that we do not significantly penalize performance of traditional clients. In this section we describe one such technique for fragmentation.

We assume that all the information needed to determine locality of access s already built into the relations.

[4]It might be unacceptable to reorganize a database every time a new access pattern is pinpointed

In our example of the Bay Area repairmen, the relation has a field indicating the city of the customer. This field can be used to determine locality of access. If such information is not available, additional fields need to be incorporated to ensure that locality of access can be determined from the extension of the relation. A *hoard key* is any such key of the relation that captures access patterns. In particular, it can be any primary or secondary key of the relation, or a hoard key of another relation referenced by this relation.

## 4.1 Logical Organization

We first describe the logical organization of relations in the database. From this we can derive the physical organization of the relations.

Associated with each hoard key is a set of *logical* fragments that define a partition of the relation. Each fragment is defined by a qualifying relation (qualifier). The qualifiers identify the tuples that belong to the fragment. These take the form of a predicate involving fields only in the hoard key. The qualifiers aggregate key values into fragments. In the absence of explicit qualifiers, each distinct key value in the domain of the hoard key is mapped onto its own fragment.

As an example, consider the two relations with the following schemas:

**CUSTOMER(CNUM, LOCATION, NAME)**
*Primary Key* is **CNUM**,
*Secondary Key* is **NAME**,
*Hoard Key* is **LOCATION**.

and

**COMPLAINTS(CID, CNUM, DESC)**
*Primary Key* is **CID**,
*Foreign Key* is **CNUM**,
*Hoard Key* is **CUSTOMER.LOCATION**.

The qualifiers for **LOCATION** are as follows:

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

Based on these two relations we can classify hoard keys into three classes by their source:

1. *Implicit:* Each time a client hoards a fragment, an implicit hoard key gets associated with it. The qualifiers associated with the hoard key are: all tuples hoarded by the client and all tuples not hoarded by the client. e.g. If a client hoards the fragment **select * from CUSTOMER where LOCATION="San Fransisco"** then the qualifiers are $LOCATION =$ *"San Fransisco"* and $LOCATION!=$*"San Fransisco"*. The server is not required to explicitly track the data hoarded by the client (though it mus keep track of which fragments the client hoarded from), but the client itself would be aware of these qualifiers. If the server does keep a track of hoarded data, applications can use implicit hoard keys to query the fragments hoarded by the clients.

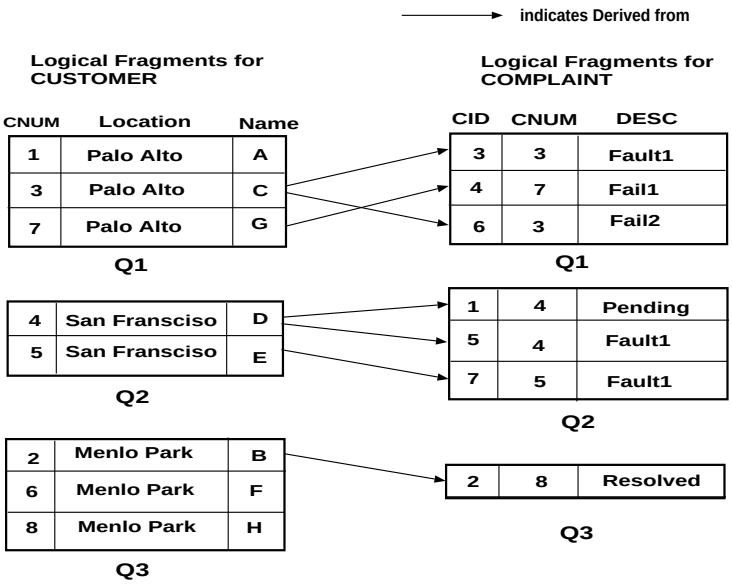


Figure 4: Sample Logical Fragmentation for **CUSTOMER** and **COMPLAINT**

Implicit hoard keys provide a way to integrate our model with other models where clients are allowed to hoard part of the relations, but where the hoard key concept does not exist.

2. *Internal:* Internal hoard keys are keys of the relation. These keys can be any primary of secondary key of the relation. As mentioned before, if sufficient information is not available to determine locality of access, the database designer must incorporate additional fields in the schema that capture the locality of access. However, we believe that sufficient information will always be available within any database. The qualifiers for the hoard keys can be explicitly specified by the database designer. The logical fragments themselves can be instantiated using a group of select statements on the entire relation. For example, if the qualifiers for the relation $R$ are $Q_i, i = 1 \cdots n$, these fragments take the form **select * from** $R$ **where** $Q_i$ $(i = 1 \cdots n)$.

   In the above example (Figure 4), the qualifiers on **LOCATION** partition the **CUSTOMER** relation into three fragments (assuming that **LOCATION** can have only one of the three values). Thus, each tuple with **LOCATION** = "Palo Alto" would be in the fragment associated with $Q_1$, etc.

3. *External or Foreign:* External or Foreign hoard keys of a relation $R'$ are internal hoard keys of a relation $R$ whose primary key $F$ is a foreign key of $R'$[5]. The qualifiers for the hoard key in $R'$ are the same as those in $R$. The relation $R'$ is logically fragmented to mirror $F$'s distribution in $R$ such that if a key value $k$ in the domain of $F$ appears in a fragment corresponding to $Q_i$ (for some $i$) in $R$ then it appears in the corresponding fragment in $R'$. The logical fragments may be specified using nested select statements as in **select * from** $R'$ **where** $F$ **in (select** $F$ **from** $R$ **where** $Q_i$**)**. A relation may have multiple foreign keys and thus potentially have more than one set of external hoard keys. It is upto the database designer/administrator to decide which of these keys to use. In our example (see figure 4), a customer with **LOCATION** = "Palo Alto" would have her complaint mapped into the

[5] The primary key requirement can be weakened further, if it can be guaranteed that each key value of the common key in $R$ is in precisely one logical fragment defined by each hoard key

"Palo Alto" logical fragment of **COMPLAINT**.

External hoard keys provide the designer with a uniform way of identifying locality of access. In our example, **COMPLAINT** does not have any intrinsic information for identifying locality of access. However, by using external hoard keys, the designer can identify and/or predict access patterns on the **COMPLAINT** relation.

The definition of external hoard keys can be extended to shared hoard keys. If both relations $R$ and $R'$ share a hoard key $H$, then the qualifiers defined for $H$ in $R$ can be inherited by $R'$. The select statement now becomes **select * from** $R'$ **where** $H$ **in (select** $H$ **from** $R$ **where** $Q_i$**)**[6]. For example, Suppose **COMPLAINT** had a **LOCATION** field. Then it suffices to specify qualifiers for **CUSTOMER** and declare **LOCATION** to be an external shared hoard key for **COMPLAINT**.

## 4.2 Physical Organization

Logical fragments merely represent partitions of relations in a database. If a relation has more than one hoard key, it would be partitioned in different ways by each hoard key, and the logical fragments from different hoard keys would overlap (though logical fragments from the same hoard key can not). Therefore, in general, these partitions can not directly mapped into disjoint physical fragments.

Whether or not the logical fragments of a particular hoard key affect the physical organization of a relation depends on whether the hoard key is a *logical hoard key* or a *physical hoard key*. If access to the logical fragments of a particular hoard key is provided using index data structures specifically built for each fragment, but the fragments are not realized on disk, then the hoard key is a logical hoard key. If each logical fragment of a hoard key is a union of one or more physical fragment of the relation on disk, then the hoard key is a physical hoard key. Physical hoard keys allow faster access to frequently hoarded data [3] by clustering data by locality of access. The logical hoard keys provide a way of clustering indices rather than data and are designed for less frequently hoarded data and for on-the-fly extensions to the database. Note that external hoard keys can be physical or logical in a relation irrespective of their status in the relation from which they have been inherited (i.e. where they are internal). In our example **LOCATION** can be a physical hoard key of **CUSTOMER** while being a logical external hoard key of **COMPLAINT**. Implicit hoard keys are by definition logical hoard keys.

If the relation has a single physical hoard key, the physical fragments of the relation are also the logical fragments of this hoard key. If there are multiple physical hoard keys, physical fragments correspond to intersections of the logical fragments associated with various physical hoard keys. This implies that each logical fragment of these hoard keys is a union of one or more physical fragments.

Logical hoard keys are useful for a database designer/administrator who wishes to incorporate new access patterns into an existing database. Since these hoard keys do not affect the physical organization of relations, they provide a efficient mechanism for creating new logical fragments. If a new physical hoard key is created or qualifiers for an existing physical hoard are modified, the corresponding relation must be reorganized on disk in a costly operation.

In general, a relation can have multiple hoard keys, some logical and some physical. There are three

[6] Note that we can not do this for most secondary keys, because the inheritance might produce overlaps

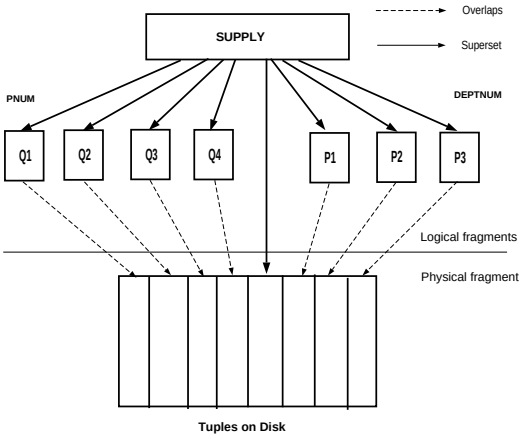


Figure 5: Both hoard keys are logical

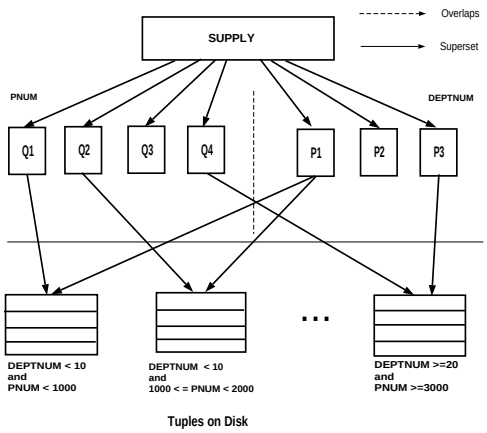


Figure 6: Both hoard keys are physical

distinct cases: 1) all the hoard keys of a relation are logical; 2) all the hoard keys of a relation are physical; and 3) some of the hoard keys of the relation are logical and some others are physical. For example, consider the following schema with two hoard keys, modeled from [6]:

**SUPPLY(SNUM,PNUM,DEPTNUM,QUAN)**
*Foreign Keys* **SNUM,PNUM and DEPTNUM**,
*External 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** $>= 20$

Also 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. Here the logical fragments defined by any one hoard key are disjoint, though logical fragments defined by different hoard keys 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.

The three cases are:

- **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 logical fragment associated with that hoard key is realized by constructing an separate index for tuples satisfying the corresponding qualifier.

For example, assume that **DEPTNUM** and **PNUM** are both logical hoard keys. This case is illustrated in Figure 5. 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 and $Q_i$'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 such indexes, 3 for **DEPTNUM** and 4 for **PNUM** .

This scheme can be used when the workload primarily consists of traditional clients with a few hoard clients. Since the physical organization of the database is unchanged and the number of hoard clients is small, traditional clients are mostly unaffected. The indexes defined on the logical fragments provide faster access to hoard 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[7]. 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 6). 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. In this case, each physical fragment also represents an update locality.

  This scheme can be used where there are large number of clients hoarding data localized by both **PNUM** and **DEPTNUM** . Also, the database designer can use the qualifiers to reduce the size of each physical fragment to an extent where *entire fragments can be hoarded by the hoard client.* Thus the database designer can eliminate indexing on hoard keys. This is because every hoard request by the client can be satisfied by dumping the entire physical fragment corresponding to the clients update locality[8].

  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 in the domain of the primary key may now appear in different fragments and would not be physically contiguous.

- **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 7. The database is physically fragmented for each $P_i$. In this case there are three fragments

[7] 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.

[8] indexing is still required to support traditional queries

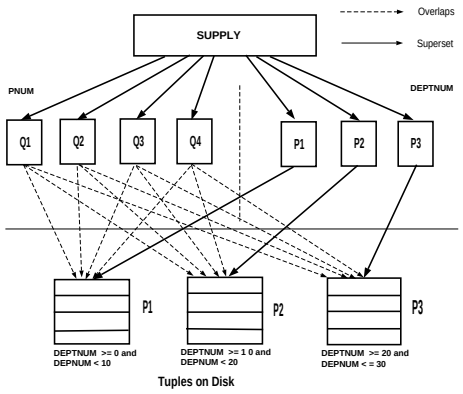


Figure 7: One of the Hoard Keys is physical

consisting of tuples with **DEPTNUM** with values in the range $[0, \ldots, 9]$, $[11, \ldots, 19]$ and $[20, \ldots, 30]$ respectively. Each of the logical fragments for the $Q_i$'s are realized using indexes.
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.

Schemes 2 and 3 also provide us with the ability to distribute physical fragments across multiple machines. Thus, each physical fragment could be made autonomous with its own server, in effect replacing the centralized server with a distributed database system.

The logical and physical fragments defined by the hoard keys can be represented by a directed acyclic graph (figure 7), whose nodes are the fragments defined by various hoard keys and the edges are the superset and overlaps relationships. We call this graph a fragment graph. At the root of the graph is the entire relation (which by itself is a logical fragment). If the relation does not have physical hoard keys the entire relation is also the physical fragment (figure 5). If there are physical hoard keys, the physical fragments are partitions of the logical fragments defined by the physical hoard keys. The logical fragments for the hoard keys appear in between the two.

In general, the server must be prepared to respond to hoard requests on any fragment defined in the fragment graph. For example, if a hoard client wishes to hoard an entire relation, the server must support this operation. Furthermore, the server must manage access control and concurrency control for fragments at each level of the graph. In practice, it would suffice to manage these issues at the level of physical fragments and to propagate updates and access requests for any logical fragment down to the physical fragments using the fragment graph.

The graph in figure 7 does not show any intermediate levels of fragments between logical fragments and physical fragments. However it is conceivable to use a similar graph to support arbitrary unions and intersections of logical fragments.

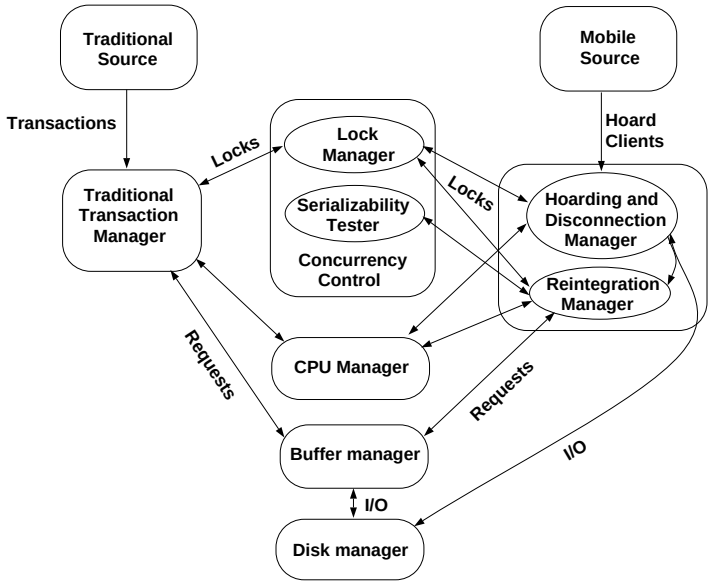


Figure 8: Simulation Block Diagram

# 5 The Simulation Model

The primary purpose of our simulation is to measure performance of both hoard and traditional queries with concurrency control, and also to determine the reintegration performance of mobile transactions (these are transactions running on a disconnected hoard client) in terms of number of transactions rolled back. Our simulation model in [3] measures disk access performance without taking into account the effects of concurrency control. This simulation demonstrated that raw access performance can be improved by a factor of 3 for hoard clients, while only marginally impacting traditional clients. The simulation model presented here additionally incorporates concurrency control.

We have developed a simulation package in the C programming language to simulate our system. Our simulation model is a multi-cpu multi-disk multi-threaded database system with pessimistic 2-phase locking on the server. The entire hoard/disconnect/reintegrate cycle is simulated for hoard clients. The simulation consists of the following components (Figure 8):

1. *Traditional Transaction Source.* The source of all traditional queries. The inter-arrival times are negative exponentially distributed.
2. *Mobile Client Source* generates hoard clients which hoard/disconnect/reintegrate. Again the inter-arrival times are negative exponentially distributed.
3. *Traditional Transaction Manager* is responsible for requesting locks, requesting I/O and committing traditional transactions. Deadlock avoidance is used to prevent deadlocks.
4. *Mobile Transaction Manager* manages the entire hoard/disconnect/reintegrate cycle for hoard clients. For each hoard client generated by the Mobile Client Source this module generates hoarding requests, disconnects the hoard clients and then performs reintegration of individual mobile transactions issued on this client. The disconnection time is negative exponentially distributed.
5. *CPU Resource Manager* keeps track of the CPU pool, allocates CPUs and ensures progress by gener-

| Parameter | Description | Value |
|---|---|---|
| *DBSize* | Number of Pages on Disk | 409600 |
| *PageSize* | Size of each page | 4096 bytes |
| *TuplesPerPage* | Number of Tuples in a Page | 40 |
| *NumDisks* | Number of Disks | 10 |
| *NumCPU* | Number of CPUs | 10 |
| *NumFragments* | Number of physical fragments | $10 - 1000$ |
| *HoardSize* | Average Number of Tuples hoarded by a Hoard client | 10000 |
| *TraditionalSize* | Average Number of Tuples accessed by a traditional transaction | 100 |
| *ProbTraditionalHot* | Probability that a traditional query is a range querie | 0.8 |
| *ProbTraditionalWrite* | Probability that a traditonal i/o request is an update | 0.2 |
| *TraditionalThinkTime* | Average think time for the traditional transaction source | 2 *seconds* |
| *HoardThinkTime* | Average think time for the hoard client source | 1 *minute* |
| *HoardDisconnectTime* | Average amount of time for which a hoarder disconnects after hoarding | 15 *minutes* |
| *BufferSize* | Number of frames in the buffer pool | 1000 |
| *AvgMobileTransactions* | Average number of transactions on the mobile that operate on the hoard | 15 |
| *MobileTransactionSize* | Average size of each mobile transaction | 10 |
| *ProbMobileWrite* | Probability of a mobile request being a write request | 0.5 |

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

ating the CPU clock ticks.

6. *Concurrency Control and Lock Manager* manages locks and also performs global serializability testing for reintegrating transactions. Locks in our model are page level locks. No assumptions are made about the number of locks obtained by a transaction on a single page or number of transactions locking a page. Transactions are allowed to upgrade read locks to write locks on a page if no other transaction has read locked the page. Global serializability testing is performed using Davidson's algorithm or by using a fragment level CODA like scheme. In the latter, if any tuple in the fragment is updated after the client hoarded data, all the transactions being reintegrated from the client are rejected and rolled back.
7. *Buffer Manager* manages the buffer cache. We use an LRU page replacement policy.
8. *Disk Manager* manages the disk pool. Each physical fragment is striped across all the disks.

Each level of the simulation communicates with upper layers using function callbacks. This allows us to run various parts of the simulation at different time granularities. In our simulations CPU manager is clocked faster than the disks. We have not attempted to explicitly quantify CPU performance. Rather, CPU performance is implicitly specified as the average number of requests the transactions can generate in one simulation cycle. The actual number of requests is distributed uniformly around this average.

## 5.1 Description of the Simulation

Our simulation model models the database as a single relation mapped onto a collection of *DBSize* pages of size *Pagesize* each. Each page holds *TuplesPerPage* tuples. Other parameters of our model are given in Table 1.

The traditional transaction source generates traditional transactions by sleeping for a random negative exponentially distributed amount of time with mean *TraditionalThinkTime* and then issuing a single traditional transaction. The transaction is "hot" with a probability *ProbTraditionalHot* (the concept of hot and cold accesses is defined in [4]). (A query is "hot" if it accesses a range of tuples. A "cold" query accesses tuples at random.) Both types of queries write to the disk with the probability *ProbTraditionalWrite*. Each transaction accesses a random uniformly distributed number of tuples with mean *TraditionalSize*. The transactions go from the source to the traditional transaction manager. The manager tries to acquire a CPU slot for the transaction. If a CPU slot is acquired the transaction manager uses the CPU time to acquire locks and then dispatch i/o requests to the buffer manager.

The mobile client source generates hoard clients by sleeping for random negative exponentially distributed time with mean *MobileThinkTime* and then generating a hoard client. Each hoard client is "hot" in the sense that it hoards from precisely one physical or logical fragment, any "cold" mobile queries are clubbed with cold traditional queries. *FracHoardLogical* of the hoard clients hoard data from a logical fragment defined by a logical hoard key. All other hoard clients hoard data from single physical fragments. As for the traditional clients the mobile client manager first attempts to acquire a CPU slot for the client. On receiving a slot the hoard client hoards a random uniformly distributed number of tuples with mean *HoardSize* from the corresponding physical or logical fragments. The mobile client manager acquires read-shared locks on the tuples to be hoarded and issues the read requests directly to the disk, effectively bypassing the buffer manager. Note that hoarding is a read-only operation. The mobile client manager then disconnects the client for a random negative exponentially distributed amount of time with mean *HoardDisconnectTime*. The disconnect operation also frees the CPU slot. When a client reconnects the mobile client manager re-acquires a CPU slot, and then reintegrates a random uniformly distributed number of transactions (Mean=*AvgMobileTransactions*), each of random uniformly distributed size (Mean=*MobileTransactionSize*). These transactions operate only on the hoarded data. The transactions write to the hoard with a probability *ProbMobileWrite*. The mobile client manager then validates each mobile transaction (using the global serializability testing routine from the concurrency control module). If the transaction is serializable, the mobile client manager writes all updated tuples, else the transaction is rolled back. The mobile client manager also takes care of cascaded rollbacks if any for those mobile transactions that have read dirty data from a rolled back transaction.

### 5.1.1 Results and Discussion

We conducted a set of simulation experiments to determine various performance measures for a sample system. Each experiment consisted of 10 runs of 4 simulated hours each. We used a database with a single relation of size 409600 pages with 40 tuples per page as our sample database. (This implies a database size of 1.6 GB for a page size of 4096 bytes). Table 1 specifies values for other parameters in our simulation.

The simulation runs fall into two broad classes. In the first class we varied only the number of physical

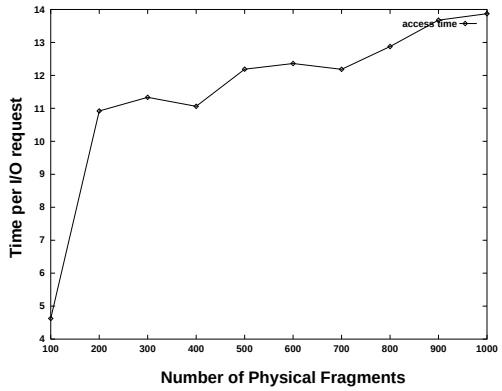


Figure 9: Time per I/O requests from traditional transactions (varying fragments)

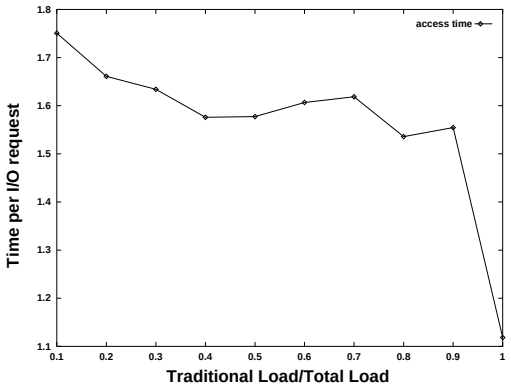


Figure 10: time per I/O requests from traditional transactions (varying loads)

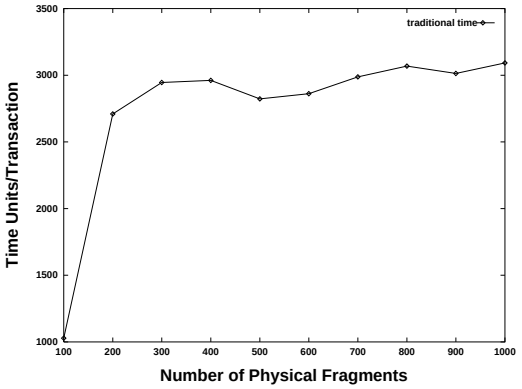


Figure 11: Time per traditional transaction (varying load)

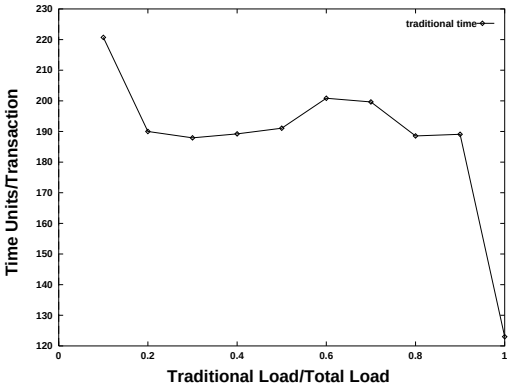


Figure 12: Time per traditional transaction (varying load)

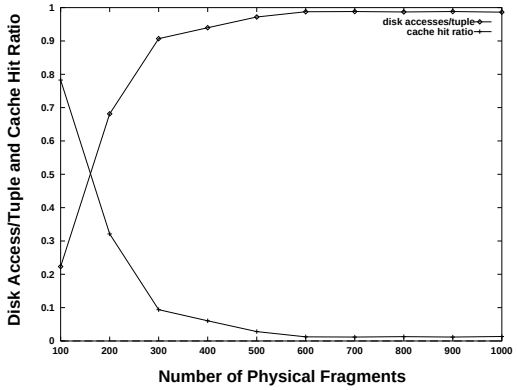


Figure 13: Cache Performance for traditional transactions (varying fragments)

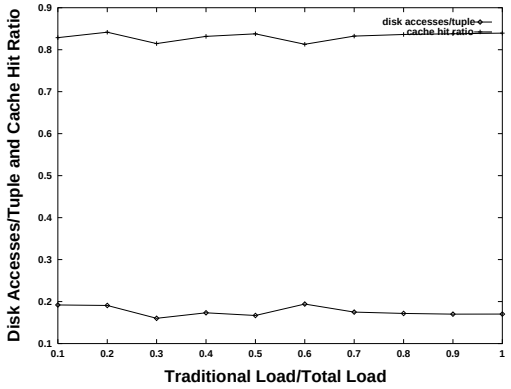


Figure 14: Cache Performance for traditional transactions (varying loads)

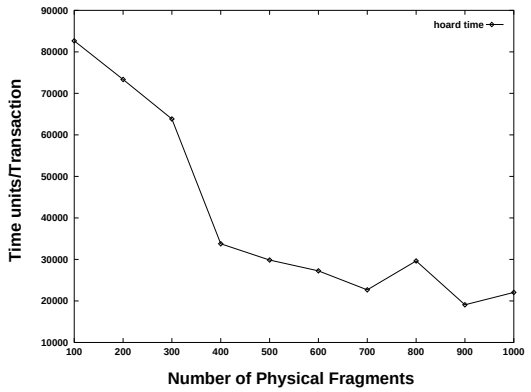


Figure 15: Time required for hoarding (varying fragments)

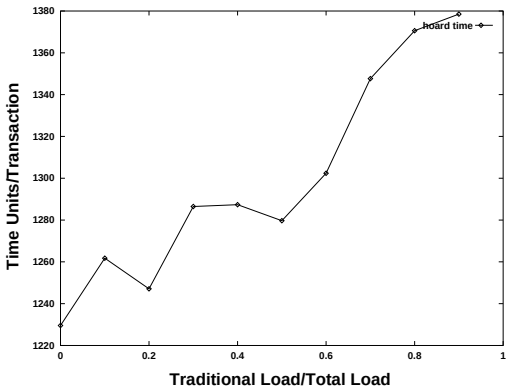


Figure 16: Time required for hoarding (varying loads)

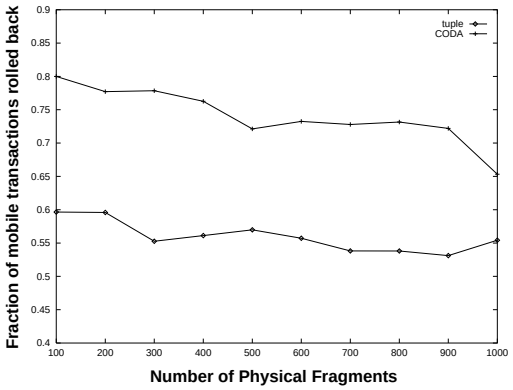


Figure 17: Reintegration performance (varying fragments)

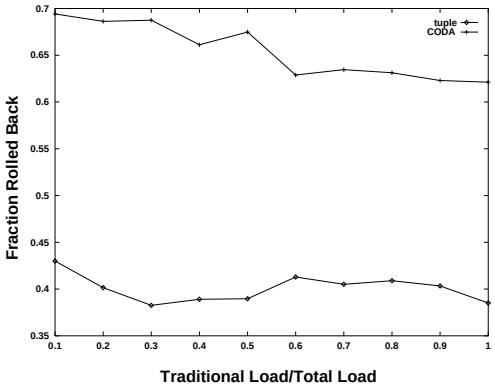


Figure 18: Reintegration performance (varying loads)

fragments in the system. In the second class of runs, we varied the fraction of the load due traditional clients on the system to total load on the system (due to both hoard clients and traditional clients). We define load as the average number of I/O requests generated per unit time. Some of the results of the simulation runs are plotted in Figures 9 through 14. Figure 9 and figure 10 plot the average number of disk accesses required to fetch a tuple from disk for a traditional transaction. Figures 15 and 16 plot the average amount of time required for a single hoard client to hoard data. Figures 17 and 18 plot the fraction of reintegrating transactions that are rolled back. We use this as a measure of reintegration performance. Figures 11 and 12 plot the average time required for traditional transactions to complete. Finally, Figures 13 and 14 plot the average turnaround time of a single request from a traditional transaction. The graphs labeled `varying fragments` have been plotted by varying the number of physical fragments of the relation. Graphs with `varying load` have been plotted by varying the fraction of the load due to traditional clients to the total load on the system.

From the simulation results, we make the following observations.

1. As the number of physical fragments increases, the performance of traditional clients degrades (figures 9, 11 and 13). This is because consecutive tuples are distributed across fragments. Thus if the number

of fragments is too large, each tuple leads to a disk access (which is indicate by the sharp drop in the cache hit ratio in figure 13).

2. As the ratio of the load offered by traditional clients to the total load offered, the performance of traditional clients improves (figures 10, 12 and 14). As indicated by flat curves in figure 14, the lack of performance performance at low load ratios is not due to cache misses or extra disk accesses, rather it is due to the fact that requests spend more time waiting to be completed as the number of hoard clients goes up (Figure graph-req-vf). Note that each hoard requests generates a big burst of read requests to the disks, potentially delaying other i/o activity. This delay effect is clearly demonstrated by the impact of varying load ratios on average time required for each traditional transaction to complete (figure 12). Here the time required is significantly lower when there are no hoard clients (i.e. traditional load/total load=1.0). We expect that this effect would be even more marked if the hoard requests do not bypass the buffer manager.
3. As the number of physical fragments goes up, hoarding performance improves (figures 15 and 17). Figure 15 indicates upto 4 times gain in performance. This is due to the clustering effect of the fragments. Moreover, reintegration performance (figure 17) improves if CODA style, fragment level reintegration is performed. This gain in performance is due to the reduction of the collision cross-section of the reintegrating transactions; which is of the order of an entire physical fragment for CODA style reintegration.
4. As the load offered by traditional clients is increased in comparison to that offered by hoard clients, there is a tremendous negative impact on the hoarding performance (figure 16). However, reintegration performance (Figure 18) only degrades a little. We think that this is due to the fact that larger amounts of data requested for hoarding remains locked by traditional transactions as the number of traditional transactions increases. Due to pessimistic 2-phase locking, the locks are not released till the transactions commit, thus delaying the hoard transactions. However such locks would not affect the number of transactions reintegrating (they would only delay the reintegrating transactions).
5. As expected, performance of the CODA style reintegration scheme is always worse than tuple level reintegration (Davidson's algorithm with granularity of conflict detection being a tuple). The CODA like scheme can never outperform tuple level reintegration since the collision cross section for each reintegrating transaction that is offered by the CODA like scheme is always a superset of that offered by tuple level reintegration. However this scheme is practicable if the number of fragments is large (figure 17). For example, in our simulation model with a 1000 physical fragments, CODA style reintegration incurs an additional penalty of only 10% over tuple level reintegration.

# 6 Conclusions and Future Work

The new architecture presented here meets our design goals (Section 1.2). In particular we believe that such an architecture is a major step towards providing full fledged database systems to mobile users. However this architecture introduces many new research problems and alternatives:

- **Performance Tuning**: There is a tradeoff between the performance of general purpose clients and hoard clients. This tradeoff to be evaluated to produce optimal performance. In particular the designer

needs to determine what sort of workload can be expected in the system, and decide how many physical hoard keys to use, how many physical fragments to create, etc.

There is also the problem of specifying the qualifying relations. These qualifiers 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 are needed (see [16]). 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.

If CODA like reintegration at fragment granularity is used (section 3.3), any write to a fragment would invalidate all clients hoarding from that fragment. To avoid this we could define penalty measures for traditional transactions attempting to write the fragment. For example if a transaction attempts to be write to a fragment which has been hoarded by a large number of hoard clients, it can be delayed till sufficient number of hoard clients have reintegrated. Penalty measures may also be of use in other circumstances.

- **Consistency and Concurrency Control**: In this paper we have provided a very simple consistency model. More work needs to be directed into this aspect of the architecture. For example, what sort of information/data structures must be maintained? How can transaction logs be used to extract sufficient information for regaining consistency? There is also a need to find better ways to reintegrate our consistency model with local concurrency control models. For example can version vectors or some form of global timestamps be used for regaining consistency? How does the local concurrency control model interact with the global consistency model[9]?

  Fragments also provide us with a mechanism to use different concurrency control and consistency models for different parts of the database. For example, a server might allow clients to use the pessimistic checkin/checkout model for low contention fragments, while enforcing shared optimistic access to others.

  As mentioned in section 3.3, it is conceivable to have consistency management at various granularities. The impact of doing so on reintegration also needs to be evaluated. This is crucial for a database serving a wide variety of hoard clients with widely varying resources.

- **Reintegration**: Efficient reintegration is a major concern, especially when large amounts of data needs 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 reintegrate 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. Note that this is especially crucial if the client has the master replica of a hoarded fragment. If the master is not returned to the server within a contracted period of time, the server can preempt the client and declare its replica to be the master.

[9] As we saw in the last section, pessimistic 2-phase concurrency control had significant effect on the hoarding transactions. We believe that optimistic methods will do better.

Even though cheap, high bandwidth links are not always available, low bandwidth, ubiquitous, albeit more expensive, links such as a cellular have higher availability. These links can be used to perform limited synchronization between the client and the server, such as locking, tuple invalidation and trickle reintegration to reintegrate low volume updates[21]. This has interesting consequences for consistency management since some parts of the hoard can become "more" consistent than others over a period of time[10].

The reintegration model presented in this paper automatically prioritizes the reintegrating clients in the order in which they reconnect. For certain applications this may not be desirable. Different ways for providing different levels of priorities to different clients need to be explored.

- **Query Miss**: Disconnected operation also brings about 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 meta-data can be used in detecting query misses (for example, for range queries). Another way of avoiding query misses is to require that all queries in disconnected mode only refer to the local hoard and not to the entire fragment or relation.
- **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 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 in this case. This is especially crucial given that we allow hoarding on the granularity of logical fragments which implies that a hoard might span multiple physical fragments and hence multiple sites.

  Strategies for reorganizing relations and for creating logical hoard keys on the fly also need to be evaluated. For example can hoard profiles of the hoard clients be used to determine new access patterns? If so how can the database automatically respond by creating new hoard keys or new logical fragments? How often, if at all, should physical fragments be changed to reorganize the relations on disk?
- **Client side issues**: Throughout this paper we have focussed on the server side issues of our architecture. The impact of our architecture on clients also needs to be studied. For example, we require that all the transactions on the client be held pending a global commit. However, this may not be practicable, especially if the client indulges in high volume activity. Thus alternatives need to be designed to avoid the information escalation problem on hoard clients. One possibility is to use the transaction logs on the hoard clients to recreate the mobile transactions on the server.

[10] 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.

The only other assumption we have made about the clients is that each hoard client supports a full fledged relational database system. These clients are "fat" in the sense that they have enough resources to do so. However many of today's mobile clients, such as PDAs, may not have such resources. We call such clients "thin" clients. There is a need to support such clients. It might be possible to incorporate these clients by creating a proxy database client on the server for each such client to handle data conversion and semantics issues. However, the details of this extension need to be worked out.

# 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] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, 13(2), Jun. 1981, pages 185–221.

[3] B. R. Badrinath and S. Phatak, Database Server Organization for Handling Mobile Clients, *Department of Computer Science Technical Report DCS-TR-324*, Rutgers University, New Jersey.

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

[5] 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.

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

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

[8] S. B. Davidson, Optimism and Consistency in Partitioned Distributed Database Systems *ACM Transactions on Database Systems*, 9(3), Sep. 1984, pages 456–481.

[9] 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 the IEEE Workshop on Mobile Computing Systems and Applications*, Dec. 1994, pages 2–7.

[10] 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.

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

[18] H. T. Kung and J. T. Robinson, On Optimistic Methods of Concurrency Control, *ACM Transactions on Database Systems*, 6(2), Jun. 1981, pages 213–226.

[19] Q. Lu and M. Satyanarayanan, Isolation-Only Transaction for Mobile Computing, *Operating Systems Review*, 28(2), May 1981, pages 81–87.

[20] 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.

[21] 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.

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

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

[24] 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.

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

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

[27] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer and B. B. Welch, Session Guarantees for Weakly Consistent Replicated Data, *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, Sept. 1994, pages 140–149.

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

[29] 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.