Low-Cost Object Declustering Strategies in Parallel Temporal Object Database Systems*

Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
noervaag@idi.ntnu.no

Abstract

In a transaction-time temporal object-oriented database system (TODB), updating an object creates a new version of the object, but the old version is still accessible. A TODB will store large amounts of data, and to provide the necessary computing power and data bandwidth, a parallel system based on a shared-nothing architecture is necessary. In order to benefit from a parallel architecture, a suitable declustering of the objects over the nodes in the system is important. In this report, we study three low-cost declustering algorithms: 1) declustering based on the hash value of the OID of the objects, 2) range partitioning, based on the timestamp of the objects, and 3) a new hybrid algorithm, where current object versions are declustered according to the hash value of the OID, and the historical versions are range partitioned based on timestamp. In contrast to many similar studies, we study the performance with a workload including both read and update operations. We show that strategy 1 and 3 are the most scalable strategies, and that the new hybrid declustering strategy is especially suitable for low update rates, for example in geographical information systems and decision support systems with support for temporal data. However, in general declustering based on the hash value of the OID of the objects has the most stable and predictable performance.

1 Introduction

In a transaction-time temporal object-oriented database system (TODB), updating an object creates a new version of the object, but the old version is still accessible. A system maintained timestamp is associated with every object version, usually the commit time of the transaction that created this version of the object.

In a TODB, every previous object version is stored, and object are in general never deleted. The result is large amounts of data, which we also want to be able to query. To provide the necessary computing power *and data bandwidth*, a parallel architecture is necessary. The shared-everything architecture which symmetric multiprocessors are based on, is not truly scalable, so our primary interest is in TODBs based on shared-nothing multicomputers. With the advent of high performance computers, and high speed networks, we expect multicomputers based on commodity workstations/servers and networks to be cost effective.

In order to benefit from a parallel architecture, a suitable declustering of the objects over the nodes in the system is important. Declustering in traditional database systems is now a quite well understood

*Revised version of IDI Technical Report 3/2000, ISSN 0802-6394.

area, but declustering in TODBs has received little attention. There are several factors that makes this issue important (and difficult!):

- Navigational accesses are common in an object database system (ODB), in contrast to relational database systems where accesses are mostly set based. In an object database system, an object is uniquely identified by an object identifier (OID), which is also used as a "key" when retrieving an object. It is important that the mapping from OID to physical location (including the node) has a low cost.
- In a temporal database, time-alignment operations, for example temporal join and temporal aggregation/grouping, are important. In a time-alignment operation, objects valid at the same time have be accessed. Time-alignment operations can be very expensive, and execution of these operations can become a bottleneck if a suitable declustering strategy is not used.

In this report, we will study declustering strategies that have a low cost, and facilitates efficient retrieval of current as well as historical objects (most recent object version versus the previous object versions). The work in this report is done in the context of the Vagabond system, a parallel TODB currently under development at NTNU [6]. When discussing the performance of the declustering strategies under different workloads, we also give examples of systems and application areas where the particular declustering strategies are suitable. It is important to note that few systems for these application areas currently supports temporal data management. However, a need for temporal data management in these areas has been identified, and should be supported in the future.

The organization of the rest of the report is as follows. In Section 2 we give an overview of related work. In Section 3 we give some introductory examples. In Section 4 we describe the system model used as the context of this report. In Section 5 we describe different declustering strategies. In Section 6 we develop analytical models for the most interesting declustering strategies, and in Section 7 we use the cost models to study how different workload parameters affect the performance using the different declustering strategies. Finally, in Section 8, we conclude the report and outline issues for further research.

2 Related Work

To reduce the query costs, optimal allocation and fragmentation is very important, but complex objects, object classes and inheritance increase the size of the solution space for the data distribution problem in an ODB. In an TODB, the aspect of time makes this problem even more difficult. Although these issues have been studied by a number of researchers, the corresponding update costs have not been studied, and experiments have been done on relatively small databases. This leaves a lot of questions unanswered. We will now give a brief overview of the most relevant work on object declustering in non-temporal ODBs and in parallel temporal database systems in general.

Non-Temporal ODBs. Declustering based on the hash value of the OID has been shown to perform well in an ODB where set based operations are common [7]. However, declustering based on hashing of the OID is not guaranteed to perform well for applications with an access pattern that is more based on pointer navigation.

To reduce the cost of pointer navigation, class wise fragmentation can be used. Chen and Su [1] describe an heuristic partitioning approach based on class wise fragmentation and allocating these classes, one or more, to each node. This is mainly interesting in databases with small class sizes and

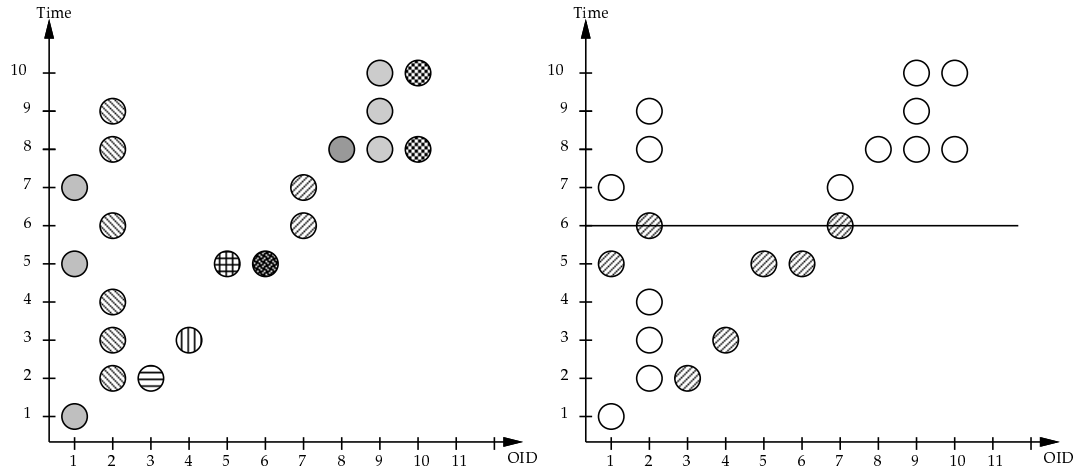


Figure 1: Object versions versus time. To the left, the circles denote object versions, and versions of the same object have the same hatching/coloring. To the right, we have illustrated a timeslice, by hatching all objects versions valid at time $T = 6$.

a large number of classes compared to the number of nodes. With a large number of objects in each class, this approach can easy give load balancing problems.

Another fragmentation approach has been proposed by Ghandeharizadeh et al. [3]. They use greedy algorithms to place all objects on the nodes in an optimal way. However, these algorithms are very workload dependent, and for these algorithms to work, 1) access statistics is needed, and 2) a costly index lookup is needed to determine on which node a particular object is located if the fragmentation should be dynamic. We also question the scalability of the algorithms. The paper says little about the CPU cost, and in the experiments reported, a very small database with only 19531 objects was used.

Parallel Temporal Database Systems. In a study of temporal query processing and optimization in multiprocessor database machines (in the context of a temporal relational database), Leung and Muntz [5] range-partitioned the tuples based on the timestamp. In a study of parallel query processing strategies for TODBs, Hyun and Su [4] used class wise fragmentation, similar to the approach used by Chen and Su [1] described above.

3 Introductory Examples

Figure 1 illustrates the evolution of objects with time. In a parallel system we want to decluster the objects over the servers. In order to avoid some kind of directory lookup when retrieving objects, the objects are declustered over the servers based on some mapping based on OID, timestamp, or both.

When retrieving an object version, we may want to retrieve the current version of an object, or to retrieve the version of an object that was valid at a particular time. If emphasis is on low cost retrieval of the current object versions, a declustering based on hashing of the OID is very suitable. However, in a transaction-time temporal ODB, we often want to operate on a snapshot of the database, i.e., on a consistent version of the database as it was on time T_i. This is called a timeslice operation, and is illustrated on Figure 1, where all object versions that was valid at time $T = 6$ are hatched.

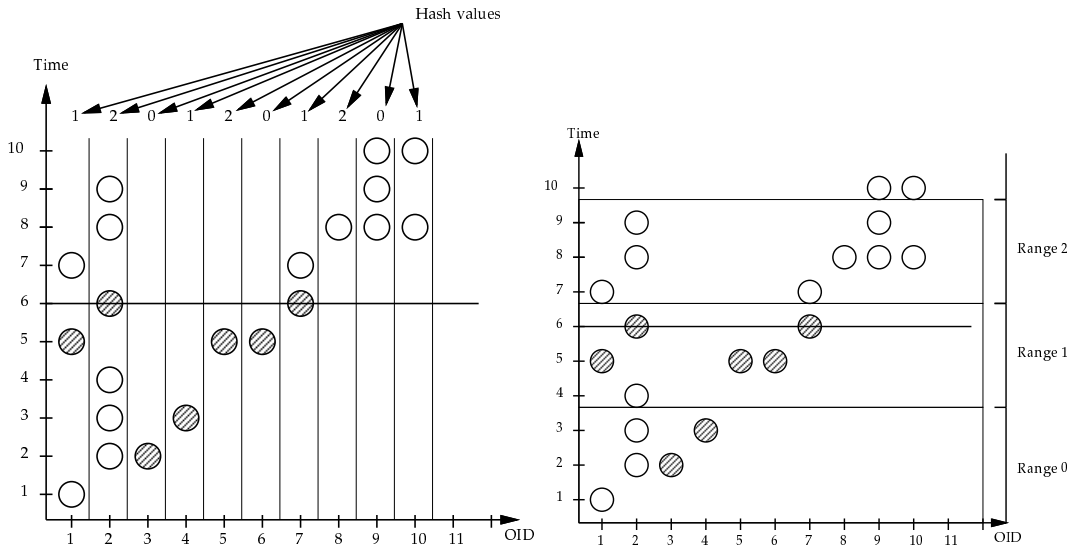


Figure 2: Two declustering strategies. To the left a declustering based on hashing of the OID, to the right a declustering based on range partitioning of the timestamp.

If communication cost is an expected bottleneck in a system, it is important to decluster the object versions in a way that minimizes communication as well as balancing the load. Two such strategies are illustrated on Figure 2.

To the left in Figure 2, a declustering based on hashing of the OID is illustrated. In this case, the hash value of the OID is used to determine which node to store an object version. This gives efficient and predictable access to all object versions, but when operating on a snapshot this approach can be less efficient. As is illustrated, object versions valid at a certain time will in general be stored on different nodes.

To the right in Figure 2, a declustering based on timestamp is illustrated. This declustering increases the probability that object versions valid in a certain time interval are stored on the same node. In the figure, we see that most of the objects valid at time $T = 6$ will be stored on the same node if this declustering strategy is used. If the (sub)transaction accessing these object versions run on the same node, we reduce the communication costs. Such a declustering is also useful for other operations that can benefit from a declustering where object versions valid at the same time are stored on the same node, for example:

- Temporal selection: Many temporal queries involves a selection on time on a collection of objects (this includes objects valid at a certain time, or objects valid in a particular time interval). If objects valid at the same time are stored at the same node, the transfer volume can be significantly reduced. This is similar to parallel selection queries in non-temporal databases.

- Temporal grouping: This is also an example where the transfer volume can be significantly reduced if objects valid at the same time are stored on the same node. In an aggregate query with temporal grouping, most of the objects in a group are already on the same node.

4 System Model

In this report we assume a database system executing on a number of nodes communicating through a message based communication network. An instance of the database system runs on each node. On each node, a number of objects is stored. In general, a client is connected to one of the nodes. Simple queries can be run on the same node as the client is collected, more complex queries are parallelized and run in parallel on all nodes.

When objects are created or updated, they are stored on one of the nodes, according to a declustering strategy. The declustering strategy is used when retrieving objects, in order to be able to know on which node a certain object or object version is stored. Each node indexes the objects stored on that node.

In addition to traditional object retrieval and object relational operations, temporal operations have to be supported. The most costly temporal operations are those requiring time-alignment of objects.

5 Object Declustering Strategies

The object declustering problem has much in common with the traditional object clustering problem. In both cases, the existence of applications with very different access patterns makes it difficult to predict which objects will be accessed together, and should be stored close to each other. Clustering in TODBs is yet "uncharted territory", and we suspect that problem will prove to be even harder. This makes us believe that instead of using large resources to try to cluster objects together, it is better to simply distribute data as evenly as possible over the nodes, in a way that simplifies retrieval of current versions and keeps down the cost of time-alignment operations. For other query types, we rely on data re-distribution during the query execution.

Declustering can be horizontal, vertical, or a combination. We will in this report concentrate on strategies for horizontal declustering, and in the rest of this section we will describe three different low cost declustering strategies: 1) declustering based on the hash value of the OID of the objects, 2) range partitioning, based on the timestamp of the objects, and 3) a new hybrid algorithm, where current versions are declustered according to the hash value of the OID, and the historical versions are range partitioned based on timestamp.

5.1 OID Based Declustering

In most ODBs, an object in a database is stored in a container/file, which can be logical or physical. A container identifier is often included in the OID, in addition to the unique number. To keep the discussion general, we consider an OID with the following attributes:

- *CONTID*: Container identifier, which identifies the container the object belongs to.
- *USN*: Unique serial number. Each object to be included in container *CONTID* gets an *USN* that is one larger than the previous *USN* allocated in the same container.

Simple OID based declustering can be based one or both of attributes of the OID. To be applicable, the strategy should decluster objects in a way that makes subsequent retrieval possible without the need for an additional (and costly) OID-to-node mapping index. We will now discuss the two strategies, and their characteristics.

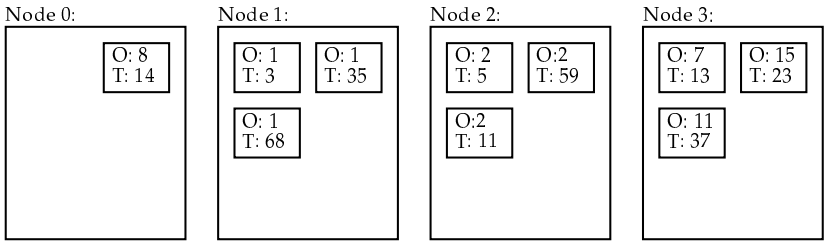


Figure 3: OID based declustering. In the figure, an version is identified by the OID O:i and the timestamp T:T_i.

5.1.1 Declustering on *OID*

When the *OID* declustering strategy is used, the node is determined by hashing the *USN*, or the combination of *USN* and *CONTID*, which has the same characteristics. In this way, objects will be evenly distributed over the nodes, and skew is unlikely to be a problem for queries that only access current versions of the objects. Figure 3 illustrates the *OID* declustering strategy in a system with $N_N = 4$ nodes. The node of an object is determined from the equation $OID\%N_N$, where % is the MOD operator.

This declustering strategy is especially applicable in the case of very large collections or sets, and queries are mostly done on the current versions of the objects. A typical example is aggregation, where a skew free initial distribution is ideal for the first phase of a parallel aggregation involving grouping, in which local aggregation is performed. In the second phase, the results from the local aggregation are redistributed, based on a hash partitioning value. However, there are two problems with this declustering strategy:

1. All versions of an object will be stored on the same node. If operations on historical versions of objects are frequent, for example historical aggregation, and the number of versions for each of the objects involved is skewed or only a small subset of the objects are accessed, we can get a load balancing problem.

2. Versions of different objects that are valid at the same time will be on different servers, making time-alignment operations expensive.

5.1.2 Declustering on *CONTID*

If only the *CONTID* is used as a parameter to the hash function, all objects that belong to a container will be stored on the same node. This also includes all the historical versions of the objects in the container. This can result in the same problems when doing queries on historical versions as when only the *USN* was used, and more important, the probability of skew is very high when all objects of an container is stored on the same node.

This declustering strategy is not as good for large sets as the *USN* only strategy. However, there are cases where declustering on the *CONTID* still can be beneficial:

- Operations on medium sized collections, where member objects are processed together One example is aggregate operations. With medium size collections, the collections are not large enough to make parallel processing (on several nodes) beneficial.

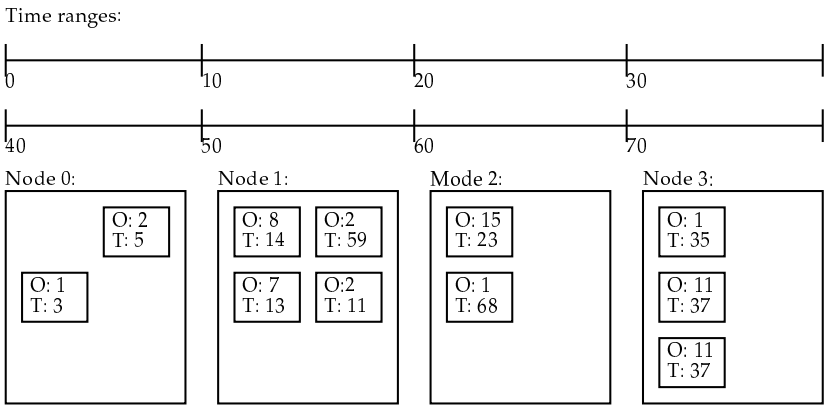


Figure 4: *TIME* declustering. On top of the figure, over the nodes, the time ranges covered by each node are illustrated. For example, node 0 covers the intervals from 0 to 9 and 40 to 49, node 1 covers the intervals from 10 to 19 and 50 to 59, etc.

- In cases where there are many references between the objects in a collection.

Because of the possible skew problem when using of *CONTID* based declustering, we do not consider this strategy as appropriate, and do not discuss it further in this report.

5.2 Timestamp Based Declustering

In a declustering strategy based on the timestamp[1] only, both hash partitioning and range partitioning can be used. If the timestamp is used as input to a hash partitioning function, we would effectively distribute *all versions of all objects* over the nodes, with little skew. However, *we would not be able to know which node that contained a particular object*, as we would in most cases not know the timestamp of a particular version (note that this problem also applies to the current versions of the objects). To retrieve an object, broadcasting would be needed. This is very costly, and is not compensated by any other benefits from declustering by hashing the timestamp.

Range partitioning based on the timestamp is a more interesting strategy. Although the problem with retrieving current versions of objects is still present, this problem is partially compensated by the increased probability of having objects with timestamp values close to each other on the same node [5], which is very beneficial when processing time-alignment operations.

When range partitioning, hereafter called *TIME* declustering, is used, the number of time ranges should be much larger than the number of nodes, and these ranges should be distributed round robin across the nodes. The reason for this, is that time has an essentially unbounded value, as it is ever increasing, and that this strategy reduces the probability of skew. If many heavily accessed objects are created close in time, many short ranges instead of a few large ones reduces the probability of skew. However, at the same time, by increasing the number of ranges, and distributing them over the nodes, we gradually loose the possible benefits in time-alignment operations, the probability of objects related in time are on the same node decreases. In this case, we have to make a tradeoff. This partitioning problem is similar to the partitioning problem in the *partitioned band join algorithm*

[1]Note that in a transaction-time temporal database, we do not know the end timestamp when we create a new version. Therefore, only the start (commit) timestamp can be used, using the end timestamp is not an option as it would be in a valid time temporal database.

proposed by DeWitt et al. [2]. DeWitt et al. used sampling methods to determine the range sizes, but in our context the problem is more difficult because the partitioning is not bound to the relatively short duration of a query. If sampling should be applicable in our context, it is likely that non-uniform range sizes should be used (at the expense of a lookup table to be able to determine which node covers a particular point in time).

Figure 4 illustrates the *TIME* declustering strategy. The node where an object version is stored is determined from the timestamp of the object. On top of the figure, we have illustrated the time ranges covered by each node. In this case, a node only covers two time ranges, but in general, the number of ranges will be higher. Note that even if an object covers (is valid) in more than one time range it is only stored once, on the node that covers its timestamp.

When declustering on timestamp, object access operations can be done as follows:

Create or update object: The object version is stored on the node determined by the commit timestamp.

Retrieve an object version valid at time T_i: When requesting the object version valid at time T_i, we first send a "get version valid at time T_i" message to the node N_i which includes the time range $R_i = [T_j, T_k>$, where $T_j \leq T_i < T_k$. An object version can cover more than one time range, and in that case it is possible that the object version was created in one of the previous time ranges. If the object version valid at time T_i was not created in the time range [T_j, T_i],[2] we have to search previ ous time ranges as follows:

1. The probe message is forwarded to the predecessor node. We "follow the timeline" when searching for the object version valid at time T_i, which means that we only consider objects created during R_{i-1} at this time. The message is forwarded until the actual object version is found, or all nodes have been probed. This can be illustrated with the following example, based on the declustering of objects on Figure 4:

   Example: Assume a search for the object with $OID = 1$ valid at time $T = 55$. We first probe node 1, but an appropriate object version is not found on that node, and we continue with node 0. The node contains an object version created at time $T = 3$, but we follow the timeline, and only consider object versions created during the time range [40, 50>. However, no object versions of object $OID = 1$ was created in that time range, so we continue with node 3 which covers the time range [30, 40>. Here we find an object version created at $T = 35$, and this is the desired result of the search.

2. One node will in general cover many time ranges. It is not necessary to forward the probe message several times through the "ring of nodes" to follow the timeline. To avoid this, we determine for each node we probe what is the most recent version of the searched object created before time T_i stored on that node. The timestamp of this version is included in the probe message. If the probe message already contains such a timestamp, the new timestamp replaces the existing timestamp if it is more recent than the previously determined most recent version. When the probe message has been through all nodes in the ring, and has reached the node which has node N_i as its predecessor, we are able to know which node stores the relevant object version. This can be illustrated by the following example, also based on Figure 4:

[2] Note that the node can also contain versions created in the time range $<T_i, T_k>$, but these versions are of no interest in this search.

Example: Assume a search for the object with $OID = 15$ valid at time $T = 75$. We start the search at node 3. However, no object version of the object with $OID = 15$ was created between [70, 75], so we have to probe the predecessor node, which is node 2. No object version of the object with $OID = 15$ was created in the time range [60, 70>, so we have to probe node 1. However, an older object version is stored on node 2, so that in the probe message to node 1 we also include the timestamp of the most recent object version stored on node 2, i.e., $T = 29$. No relevant object version is found on node 1, and the probe message is forwarded to node 0. This is the last node to probe. It contains no object version of the object with $OID = 15$, so that at this time, we know that the appropriate object version is the one with timestamp $T = 29$. A request message is sent to node 2 where this version is stored, and node 2 sends the actual object version to the requesting node.

3. It is also possible that the object was not yet created at time T_i. If this is the case, it can be determined from the probe message, where the timestamp of the most recent object version as described above will not have been set.

Retrieve the current version of an object: We do not know anything about the timestamp of the current version, so we have to probe all nodes to determine which node has the most recent version. This can be done in a number of ways:

1. First send a "get timestamp of most recent version of object i" message to all nodes, and then retrieve the object version from the node containing the most recent version.

2. Send a "get most recent version of object i" message to all nodes. All nodes return their most recent version of object i. In this way, we avoid some delay, but use more of the communication bandwidth.

3. Every time we create a new current version, we send a message about this to the node where the previous current version was stored (in general, when an object is updated, we know the timestamp, and implicitly the node, of the previous version). When we want to retrieve the current version of an object, we send a "get current version" message to all nodes. The node containing the current version knows this, so that only this node has to send an object back to the requesting node. We consider this to be the best strategy, as it keeps both delay and use of communication bandwidth to a minimum. This will compensate for the increased update cost.

A better alternative, especially if efficient broadcast is not supported, is to consider the retrieval of an current object version as the retrieval of an object version valid at time *now*. Assuming that most read current accesses are to the most heavily updated objects, this strategy will have a lower cost than involving all the nodes in the object retrieval as is the case when broadcasting is used. The drawback of such an approach is a possibly increased latency.

5.2.1 Possible Problems with *TIME* Declustering

The main problem with range partitioning is the same as with declustering on the hash value of the timestamp, we still do not know which node contains an object with a particular OID. Therefore, this strategy is only useful if timeslice operations are frequent compared to object navigation. Another problem in the context of a transaction-time database, is that all updates at a given time are done on one node, the node which contains the $[T_j, NOW>$ time interval.[3] In a system with high update

[3] NOW is the current time.

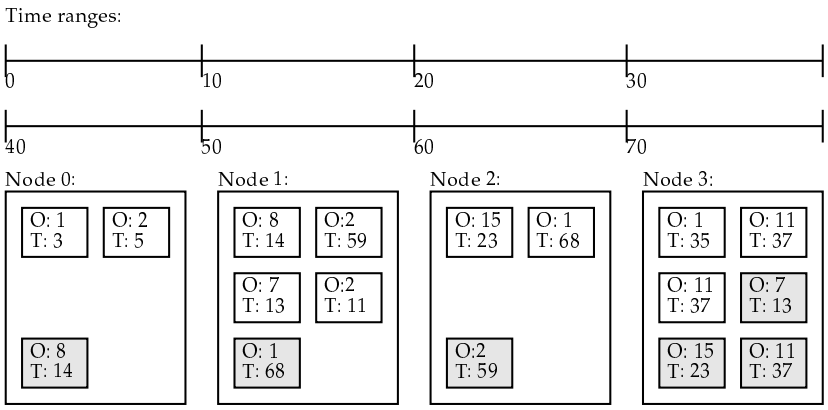


Figure 5: *OID-TIME* declustering. The replicated current version objects are illustrated by hatched boxes in the figure.

rates, this node can become a bottleneck. The fact that many of the requests for current versions of the object will be satisfied by this node as well makes this problem very serious.

To reduce the cost of subsequent timeslice operations, it is possible to store an object version on all nodes whose timestamp range is (partially) overlapped with the time the actual object version was valid. This increases the storage cost, but by using a reasonable size of the timestamp range, the amount of replication does not have to be too high. However, every time we start on a new time range, i.e., writing to a new node, all objects that are still current have to be written to this node (but note that an object version will only be written once to each node, so that in a system with N_N nodes, an object version that is very infrequently updated will be rewritten only during the first $(N_N - 1)$ time ranges after the version has first been written).

Replication is a very costly operation, and implies that timestamp based declustering with replication is only beneficial if most objects are either updated very often or very seldom. In the first case, the object version is only written once, while in the other case, it will be written N_N time, on each node, but after it has been stored on all nodes, it will not incur any further replication cost. However, the storage costs in a database where data is not deleted is already very high, so we expect the additional storage costs from using replication to be unacceptable, and do not consider the use of replication in the analysis in this thesis.

Retrieving the history of a particular object can be more expensive with *TIME* declustering than with *OID* declustering, because the versions will be stored on different nodes. However, storing all versions on the same node can result in skew. Which strategy is best in this case, depends on what kind of operations will be subsequently applied to these versions.

5.3 Hybrid *OID-TIME* Declustering

We have now discussed declustering based on one of the attributes of the OID or the timestamp, and have seen that each of these strategies have both advantages and shortcomings. We will now outline a more suitable declustering strategy, the *OID-TIME* strategy, which aims at a skew free distribution, facilitates simple retrieval of the current version of objects, and keeps the costs of time-alignment operations low.

The *OID-TIME* strategy is based on replication of the current version. When an object is created

or modified, the new current version is stored both on the node determined by hashing the *OID*, and on the node determined by the range partitioned timestamp value. Retrieving the current version of an object is done by accessing the node determined by hashing the *OID*, and retrieval of a version valid at time T_i is done in the same way as with *TIME* declustering.

Figure 5 illustrates the *OID-TIME* declustering strategy. The node of an object version is determined from the timestamp of the object. In addition, the current version is stored on the node determined from the hash value of the OID. Note that when a new current version is created, the previous current version is not accessible on the node determined from the hash value of the OID, only on the node determined from the timestamp.

The *OID-TIME* declustering strategy carries an increased create and update cost, compared to the *OID* declustering strategy. To compensate for this cost, the amount of time-alignment operations has to be sufficiently high. Similar to the timestamp based declustering, all updates at a given time are done on one node. This means that this strategy is most suitable in systems with low or moderate update rates.

In a temporal database system, the database is partitioned, with current objects in the *current database*, and the previous versions in the other partition, in the *historical database*. When an object is updated, the previous version is first moved to the historical database, before the new version is stored in-place in the current database. When using *OID-TIME* declustering, the current versions declustered on OID will typically be stored in the current database, while the version declustered based on *TIME* will typical be stored in the historical database on that node. This implies than we do not need to move an object version when an object is updated, the previous current version is already stored in the historical database. When *OID* declustering is used, we need to move an object version every time an object is updated. This is an important point to note. In the cost analysis in this report we only consider inter-node communication, and not the cost of storing the objects in the nodes. Thus, in practice, the *OID-TIME* strategy will have a slightly better performance than the analysis shows.

Similar to *TIME* declustering, replication can be used in *OID-TIME* declustering as well. However, similar to *TIME* declustering, we expect the storage costs to be unacceptable high.

6 Cost Model

The goal of our analysis is to compare the different declustering strategies in a system with N_N nodes, and investigate under which conditions they should be used. We use a qualitative approach, where our goal is to achieve a cost model of the bottleneck in such a system. We do the following assumptions:

1. In a balanced system, the CPU and disk costs will be the same for the different declustering strategies, hence, we only consider the communication costs.

2. We assume a high enough degree of intra- and inter transaction processing to be able to exclude reponse times in the processing from the cost model.

3. We assume that a number of messages and objects can be packed together in each packet, so that the number of messages in itself can be ignored.

Under these assumptions, the transported data volume of the objects and messages can be used as a measure. We denote the cost of sending an object (including the object as well as object identifying information), in terms of bytes from the communication bandwidth, as C_S, and the of sending a message, for example a message requesting an object, as C_P. The parameters and functions used in the cost model is summarized in Table 1.

Parameter	Definition	Default Value
α	Fraction of accesses to hot spot objects	0.95
β	Fraction of total number of objects that are hot spot objects	0.05
C_P	Cost of sending an *OID/TIME* object probe message	16 B
C_{PP}	Cost of sending an *OID/TIME/TIME* object probe message	24 B
C_S	Cost of sending an object (including overhead)	256 B
N_N	Number of nodes	$1 \ldots 32$ nodes
P_{write}	Object write probability	0.2
P_{new}	Probability that a write creates a new object	0.2
P_{RC}	Probability that a read is for the current version	0.7
P_{RH}	Probability that a read is for a historical version	0.1
P_{TS}	Probability that a read is a timeslice operation	0.2

Function	Definition
C	Average cost of an object access
C_C	The average cost of an object create operation
C_R	The average cost of a read operation
C_{RC}	The average cost of reading a current object version
C_{RH}	The average cost of reading a historical object version
C_{TS}	The average cost of a timeslice read
C_U	The average cost of an update operation
C_W	The average cost of a write operation
P_{RU}	Probability that an object has been updated during the current time range

Table 1: Summary of system parameters and functions.

Not all the objects in a TODB are temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the objects can be defined as non-temporal, and old versions of these are not kept. Access and declustering of these objects are independent from temporal objects, and in this analysis we only consider access to and declustering of the temporal objects.

6.1 Workload Model

The workload consists of object updates and read accesses. In order to study the performance of the different declustering strategies, it is important to study the costs of the updates and read accesses together. In the modeled workload, P_{write} of the accesses are object updates, and of these, P_{new} are creations of new objects. $(1 - P_{write})$ of the accesses are accesses caused by read accesses/read queries. Denoting the average write and read costs as C_W and C_R, we calculate the cost as the average bandwidth needed for each object access operation, i.e.:

$$C = P_{write}C_W + (1 - P_{write})C_R$$

The average cost of a write is the weighted sum of the average object create cost C_C and the average object update cost C_U:

$$C_W = P_{new}C_C + (1 - P_{new})C_U$$

The read pattern from different applications can be divided into several categories, for example:

- Applications only accessing current versions.
- Applications that mostly accessing current version, and a few historical versions through navigations.
- Applications that do timeslice operations, i.e. operating on a snapshot valid at time T_i.
- Set accesses (queries) only accessing current versions.
- Set based accesses involving temporal operators.

The object accesses from these different application categories can be incorporated into 3 read classes, each representing a certain fraction P_i of the read accesses, where the invariant $P_{RC} + P_{RH} + P_{TS} = 1.0$ should be true:

1. P_{RC}: Read the current version of an object. The cost of this operation is C_{RC}.
2. P_{RH}: Read a historical version of an object, i.e., an object version valid at a particular time T_i. This category of read operations also includes the case when we want to retrieve all object versions of a particular object. The cost of this operation is C_{RH}.
3. P_{TS}: Timeslice read operations, which can benefit from a declustering strategy that aims at storing object versions valid at the same time on the same node (see Section 3). The cost of this operation is C_{TS}.

The average cost of a read is:

$$C_R = P_{RC}C_{RC} + P_{RH}C_{RH} + P_{TS}C_{TS}$$

We will now present the cost models for the *OID*, *TIME*, and *OID-TIME* declustering strategies.

6.2 *OID* Declustering

When an object is created, the node where it is to be stored is determined from applying a hash function to the OID. With a probability of $\frac{1}{N_N}$, this is the same node as the creating node. If it is another node, which has the probability $(1 - \frac{1}{N_N})$, the new object has to be sent to the actual node. This also applies to object updates. The average object create and update costs are:

$$C_C = C_U = (1 - \frac{1}{N_N})C_S$$

All versions of an object resides on the same node, so that the costs of reading current and historical versions as well as doing a timeslice read are the same. If the requested object version is not stored on the requesting node (the node which does the operation), the average access cost is the sum of the cost of the object retrieve message and the cost of returning the object:

$$C_{RC} = C_{RH} = C_{TS} = (1 - \frac{1}{N_N})(C_P + C_S)$$

6.3 *TIME* Declustering

When an object is created, the node where it is to be stored is determined from the value of its timestamp. With a probability of $\frac{1}{N_N}$, that is the same node as the creating node, and no communication is necessary. If it is another node, which has the probability $(1 - \frac{1}{N_N})$, the new object has to be sent to the actual node. The average create cost is:

$$C_C = (1 - \frac{1}{N_N})C_S$$

When an object is updated, the new object version is sent to the node determined from the timestamp. In addition, a message is sent to the node where the previous current version was stored, so that the end timestamp can be set for the previous current version (this reduces the current version access cost):

$$C_U = (1 - \frac{1}{N_N})C_S + (1 - \frac{1}{N_N})C_P$$

To read the current version of an object can be a very expensive operation when *TIME* declustering is used. All nodes have to be probed if the current version is not stored on the requesting node. The cost is the sum of sending a probe message to all other nodes, and sending the requested object back to the requesting node (the node that stores the current version can determine this from the end timestamp which is not set in the current version):

$$C_{RC} = (1 - \frac{1}{N_N})((N_N - 1)C_P + C_S)$$

An object is valid in a given time interval, from the value of the timestamp, and until the timestamp of the next object version. This interval covers one or more ranges in the range partitioning, including one or more nodes.

We assume an access pattern with a small number of frequently updated objects, where α of the updates are done to these objects. We assume that the frequently accessed objects are updated often enough to be updated more than once during each time range. The infrequently updated objects

are updated very seldom, so that each version covers more than N_N time ranges. This is a simplification, but the following example shows that it is not unreasonable: Consider a database in a stable condition, with size $S_{DB} = 32$ GB of data and average object size $S_{obj} = 256$ bytes, giving $N_{ver} = S_{DB}/S_{obj} = 128$M object versions. If we assume the number of time ranges to be $N_R = 128$, the number of object versions in each time range is $N_{objver}/N_R = 1$M versions. During each time range, $P_{new} = 0.2$ of the object versions are new objects, while $(1 - P_{new}) = 0.8$ are new object versions of existing objects. The number of distinct objects (i.e., distinct OIDs) in the database when in a particular time range i is approximately $N_d = P_{new} N_{ver} \frac{i}{128}$ objects, on average $P_{new} N_{ver} \frac{1}{2} = 12.8$M objects. Under the assumptions above, $\alpha = 0.95$ of the updates are done to $\beta = 0.05$ of the N_d objects, this means that α of the updates are done to $\beta N_d = 0.64$M objects, i.e., ≈ 1.5 updates of each hot spot object in each time range.

When requesting the version of an object that was valid at time T_i, we first send a probe message to the node which stores objects with timestamps in the time range which includes T_i (with a probability of $\frac{1}{N_N}$ this is the requesting node, and it is not necessary to send the probe message). If we assume the read pattern is equal to the write pattern,[4] the probability that the requested object version is stored on this node is $> \alpha$. Given a particular time in a time range, the probability that one of these objects has already been updated in this time range is 0.5, and with a probability of 0.5 we have to probe the previous node. The average cost of retrieving one of these object versions is:

$$\begin{array}{rcll} C'_{RH} & = & (1 - \frac{1}{N_N})C_P & \text{Send probe message} \\ & & +0.5(1 - \frac{1}{N_N})C_S & \text{Return object} \\ & & +0.5(C_{PP} + (1 - \frac{1}{N_N})C_S) & \text{Forward probe message} \\ & & & \text{which returns object} \end{array}$$

We make a pessimistic assumption about the less frequently updated objects, and assume the average time between each update is larger than N_N time ranges. Thus, the probe message has to be forwarded $(N_N - 1)$ times. The average cost of retrieving a historical object version of one of the infrequently updated objects is:

$$\begin{array}{rcll} C''_{RH} & = & (1 - \frac{1}{N_N})C_P & \text{Send probe message} \\ & & +(N_N - 1)C_{PP} & \text{Forward probe messages}[5] \\ & & +((1 - \frac{1}{N_N})C_P + (1 - \frac{1}{N_N})C_S) & \text{Return object} \end{array}$$

The average cost of retrieving a historical object version is:

$$C_{RH} = \alpha C'_{RH} + (1 - \alpha) C''_{RH}$$

The timeslice read is essentially a read of a historical version on the node that covers the actual timeslice. However, we do not need to send the first probe message because we are already on the actual node covering the timeslice time, and if the object is stored on this node we do not have to send it:

$$\begin{array}{rcll} C'_{TS} & = & 0.5(C_{PP} + (1 - \frac{1}{N_N})C_S) & \text{Forward probe message} \\ & & & \text{and return object} \end{array}$$

[4] As will be discussed later in this report, this assumption is possibly too optimistic.

[5] Here we assume that all forward probe messages include *OID/TIME/TIME*. However, if the number of object versions of each of these objects is low, it is possible to reduce the communication cost by only including *OID/TIME* in the messages until the first node with a version of the object is reached.

$$
\begin{array}{rcll}
C''_{TS} & = & +(N_N - 1)C_{PP} & \text{Forward probe messages} \\
 & & +((1 - \frac{1}{N_N})C_P + (1 - \frac{1}{N_N})C_S) & \text{Return object}
\end{array}
$$

$$C_{TS} = \alpha C'_{TS} + (1 - \alpha)C''_{TS}$$

As described in Section 5.2, using the same algorithm for retrieving current versions as historical versions can be beneficial. In that case we also avoid the need for maintaining the end timestamp, and the update cost is reduced to:

$$C'_U = (1 - \frac{1}{N_N})C_S$$

In the rest of this report, we will denote the approach where we use the same algorithm for retrieving current versions as historical versions as *TIME2* declustering.

6.4 *OID-TIME* Declustering

When creating and updating an object, the object is sent to the node where it is to be stored, based on the hash value of the OID. In addition, the object is sent it to the node where it is to be stored based on the timestamp:

$$C_C = C_U = 2(1 - \frac{1}{N_N})C_S$$

When reading the current version of an object, we can avoid communication in two cases:

- If the requesting node is the same node as determined by hashing the OID, the current version is stored on the requesting node. The probability for this is $\frac{1}{N_N}$.
- The current version of an object is also stored on the node determined by its timestamp. If the requesting node is the one that covers the current timerange *and* the requested object was written during this time range. In that case, we know that there can not exists a more recent version on another node. The probability for this is $\frac{P_{RU}}{N_N}$, where $P_{RU} \approx \frac{\alpha}{2}$ is the probability that an object has been updated during the current time range. Note that if the current version of an object was written during a previous time range covered by the requesting node, we do not have enough information to know that this is still the current version the object. The reason for this, is that when *OID-TIME* declustering is used, we do not maintain the end timestamps for the objects.

In all other cases, communication is necessary. The average cost of reading the current version is:

$$C_{RC} = (1 - \frac{1}{N_N})(1 - \frac{P_{RU}}{N_N})(C_P + C_S)$$

The cost of retrieving a historical object version and the cost of a timeslice read is the same as when using *TIME* declustering.[6]

[6]It is possible that the current version of an object is the matching object in the case of retrieval of an historical object version or timeslice, but the probability of this is low enough to ignore.

6.5 Broadcasting

Efficient broadcasting/multicasting is often supported by the communication network. If this is the case, the cost of sending the same message to all the nodes can be significantly less than the cost of sending $_N$ messages. This can be utilized to reduce the number of probe messages when reading the current version using *TIME* declustering.

Broadcasting can also be used when reading historical versions and doing timeslice read. However, this implies that all machines have to participate in every historical read/timeslice read operation. A parallel database system where all the nodes much of the time perform the same operation is not very cost efficient, and we do not consider this a good solution.

7 Analysis

The value of the parameters used in the cost model will be different from database to database, and even between different applications accessing a particular database. We will in this section study under which conditions the different declustering strategies are most beneficial, using a wide range of workloads and number of nodes.

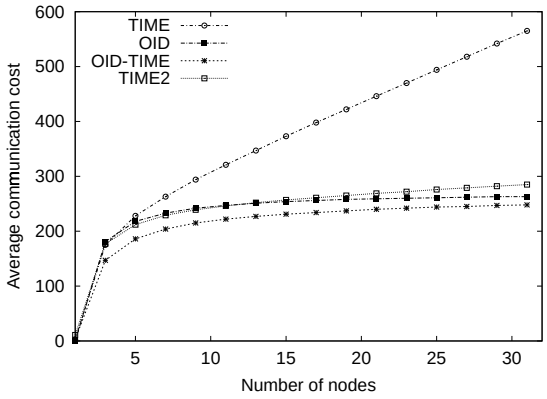
The total communication cost increases with an increasing number of nodes, because the probability of finding an object on the same node decreases. In this analysis, we study the cost with different number of nodes, and the declustering goal is to minimize the communication costs given a certain number of nodes. We also want to study the scalability of the declustering strategies, i.e., that the increase in communication cost with increasing number of nodes is acceptable.

The Effect of Different Update Rates. Figure 6 illustrates the cost with different update rates. We see that the *OID-TIME* declustering is most suitable in the case of databases with low update rates. Typical examples of applications where this occurs, are GIS (geographical information systems) and DSS (decision support systems). The cost of *TIME* declustering is high, because of the cost of retrieving current versions. The cost of *OID* declustering is low, predictable, and stable.

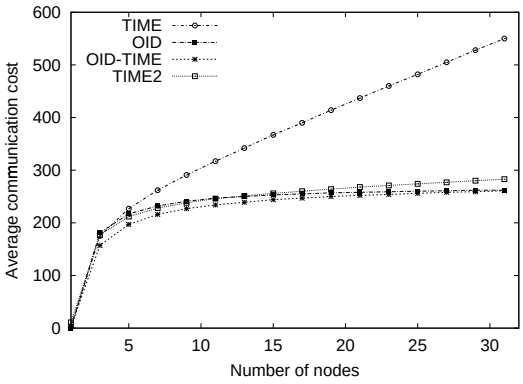
The Effect of Different Object Sizes. Different object sizes obviously affect the communication costs, and Figure 7 illustrates the cost with different object sizes. The performance of *OID-TIME* declustering gets worse with increasing object sizes because of the replication of the current object versions. The cost of using *TIME* and *TIME2* declustering is significantly lower than *OID* and *OID-TIME* declustering in the case of larger objects. The reason for this, is that the cost of the probe messages is less significant. However, it will not scale to a larger number of nodes. If the number of nodes is increased, this increases the number of probe messages and the total cost of *TIME* and *TIME2* declustering.

The Effect of Different Read Mix. Figure 8 illustrates the cost with different read mixes. We see how the performance of the *OID-TIME* and *TIME2* declustering increases relative to *OID* declustering with a larger amount of timeslice read operations. However, the amount of timeslice read operations needed to outperform the *OID* declustering is higher than what can be expected to occur in practice.

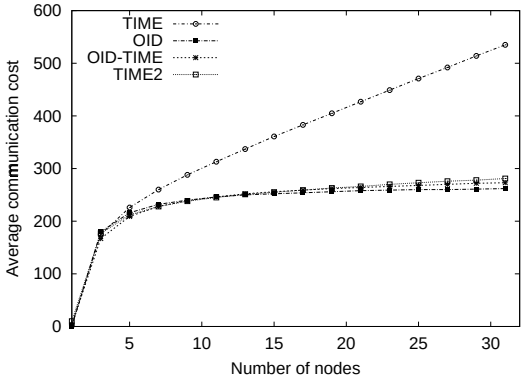
Scalability. From the figures, we have seen that the *TIME* declustering does not scale as well as *OID* and *OID-TIME* declustering. The main reason for this is that the navigational accesses to the current



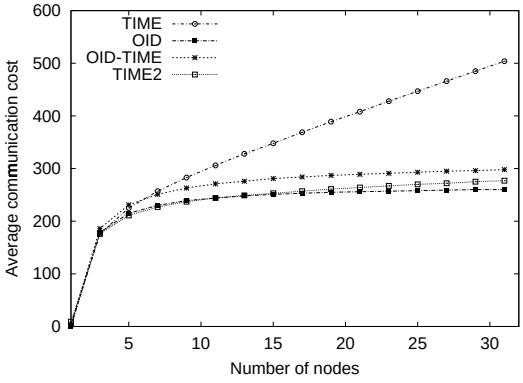
(a) $P_{write} = 0.0$.



(b) $P_{write} = 0.05$.

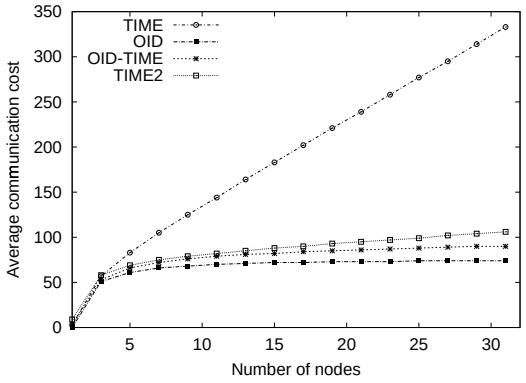


(c) $P_{write} = 0.1$.

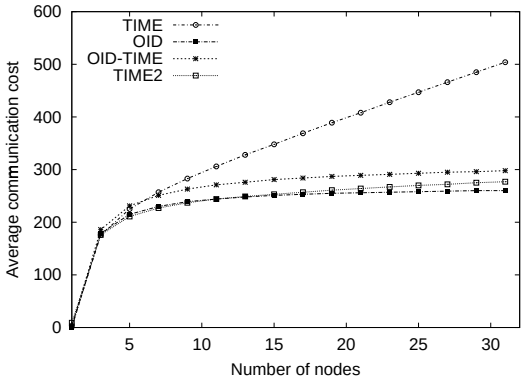


(d) $P_{write} = 0.2$.

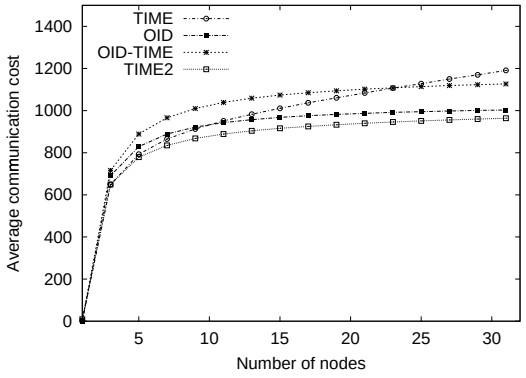
Figure 6: Cost with different update rates.



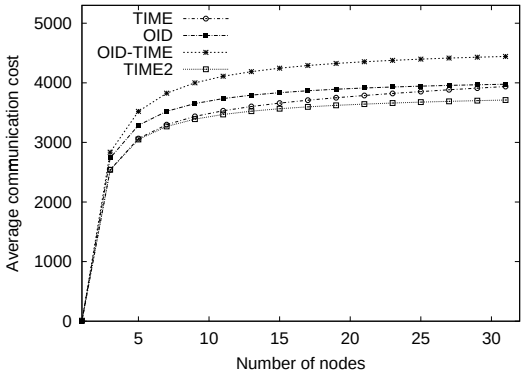
(a) $C_S = 64$.



(b) $C_S = 256$.

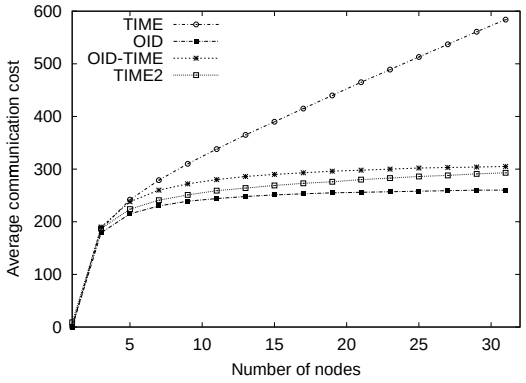


(c) $C_S = 1024$.

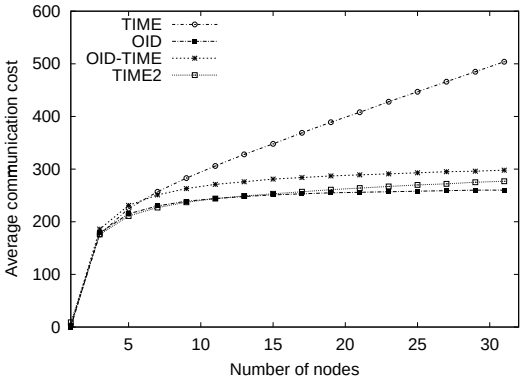


(d) $C_S = 4096$.

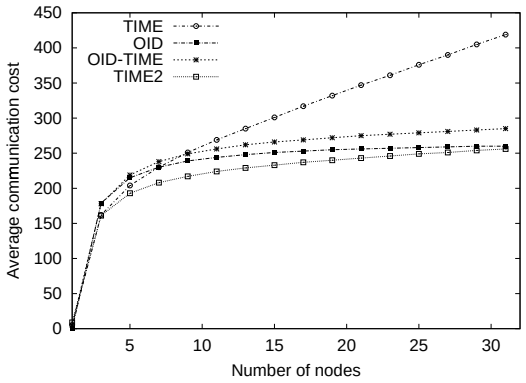
Figure 7: Cost with different object sizes.



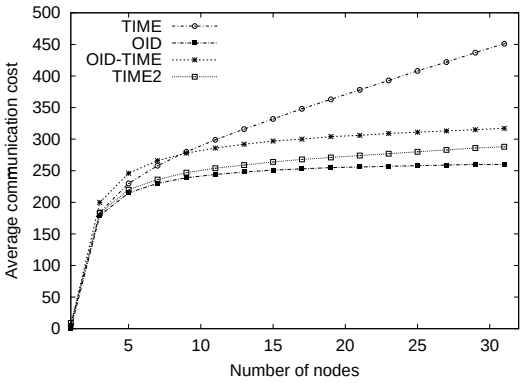
(a) $P_{RC} = 0.9$, $P_{RH} = 0.05$, and $P_{TS} = 0.05$.



(b) $P_{RC} = 0.7$, $P_{RH} = 0.1$, and $P_{TS} = 0.2$ (default mix used in the rest of the analysis).



(c) $P_{RC} = 0.5$, $P_{RH} = 0.1$, and $P_{TS} = 0.4$.



(d) $P_{RC} = 0.5$, $P_{RH} = 0.4$, and $P_{TS} = 0.1$.

Figure 8: Cost with different read mix.

version objects are very expensive. Even with small values of P_{RC} this declustering strategy performs worse than the *OID-TIME* strategy.

In the figures in this report, we have only illustrated the cost with up to 32 nodes. With a larger number of nodes, the cost of using *TIME2* declustering also becomes worse. A larger number of nodes increases the number of time the probe messages have to be forwarded. This also increases the response time, also an important issue. A larger number of nodes also makes it necessary to use more time ranges in order to avoid skew. This is likely to further increase the cost of using *TIME* and *TIME2* declustering because it increases the probability of having to forward probe messages.

8 Conclusions and Future Work

The object declustering problem has much in common with the traditional object clustering problem. In both cases, the existence of applications with very different access patterns makes it difficult to predict which objects will be accessed together, and should be stored close to each other. This makes us believe that instead of using large resources to try to cluster objects together, it is better to simply distribute data as evenly as possible on the nodes in the cluster, in a way that simplifies retrieval of current versions and keeps down the cost of time-alignment operations. For other query types, we rely on data re-distribution during the query execution.

We have in this report analyzed strategies for declustering objects in parallel temporal ODBs. Cost models were developed and used to study the characteristics of the declustering strategies, and under which conditions the different declustering strategies are most beneficial. The results, show that:

1. In a parallel ODB, the *TIME* and *TIME2* declustering strategies are in general not suitable for primary declustering (but this does not rule out these strategies as a part of the later stages in the query processing).

2. In systems with mixed workloads, the difference in performance between the *OID* and *OID-TIME* strategies is not large, but in our choice of parameter values, *OID* declustering outperforms *OID-TIME* more often than the opposite.

3. Even in systems with a high degree of temporal operations, the cost of object traversal will be the most important cost.

It is also likely that the assumptions we made for the *TIME*, *TIME2* and *OID-TIME* declustering strategies were too optimistic:

1. The read pattern does not necessarily equal the write pattern.

2. Except some systems with mostly static data, for example data warehousing systems, a much larger number of time ranges will be used.

3. A low value of β were implicitly assumed. A higher value of β would imply that each time range has to be larger, or a lower number of time ranges in total. In a more accurate model, this fact should be included in the model.

If any of these assumptions does not hold, the result will be a higher cost if using the *TIME*, *TIME2* or *OID-TIME* declustering strategies.

The final conclusion is that similar to traditional database systems, an hash based declustering (*OID* based declustering) should be used in parallel temporal ODBs as well. However, we believe that

the use of *OID-TIME* declustering during query execution is interesting, and this should be studied further.

References

[1] Y.-H. Chen and S. Y. W. Su. Implementation and evaluation of parallel query processing algorithms and data partitioning heuristics in object-oriented databases. *Distributed and Parallel Databases*, 4(2), 1996.

[2] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.

[3] S. Ghandeharizadeh, D. Wilhite, K. Lin, and X.Zhao. Object placement in parallel object-oriented database systems. In *Proceedings of the 10th International Conference on Data Engineering*, 1994.

[4] S. J. Hyun and S. Y. Su. Parallel query processing strategies for object-oriented temporal databases. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, 1996.

[5] T. Y. C. Leung and R. R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *Proceedings of the 18th VLDB Conference*, 1992.

[6] K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.

[7] J. L. Pfaltz, R. F. Haddleton, and J. C. French. Scalable, parallel, scientific databases. In *Proceedings of SSDBM'98*, 1998.