

Building a Large Location Table to Find Replicas of Physics Objects

Koen Holtman^{a,b} Heinz Stockinger^a

^a*CERN, EP division, CH - 1211 Geneva 23, Switzerland*

^b*California Institute of Technology, Mail Code 256-48, 1200 E. California Blvd., Pasadena, CA 91125, U.S.A.*

Abstract

The problem of building a large location table for physics objects occurs within a number of planned physics data management systems, like those that control reclustering and wide-area replication. To satisfy their efficiency goals, these systems have to make local or remote replicas of individual physics objects, which contain raw or reconstructed data for a single event, rather than replicas of large run or ntuple files. This replication implies the use of a table to resolve the logical, location independent object descriptor into a physical location where an object replica can be found. For modern physics experiments the table needs to scale to at least some 10^{10} objects. We argue that such a table can be efficiently implemented by limiting the freedom of lookup operations, and by exploiting some specific properties of the physics data model. One specific viable implementation is discussed.

Key words: Object location table, object-oriented databases, object clustering, object re-clustering, data replication

1 Introduction

The CMS experiment plans to store 1 Petabyte of data per year in an object database system, which will consist of 10^9 events, beginning in 2005. A single event will have at least some 10 objects associated with it, each individual object holding some raw or reconstructed data for the event. This means that at least 10^{10} objects are created annually. Each of these objects can potentially be replicated to a Regional Centre. Multiple replicas of an object can also exist at a local site when objects are reclustered or staged from tape to disk. In both of these cases, an object location table (OLT) is required to resolve the logical, location independent object descriptor into a physical location where an object replica can be found.

1.1 Logical and physical object identifiers

In an object-oriented database management system, each single object in the database is identified uniquely by a so called object identifier (OID) which can either be physical or logical. Logical OIDs are preferred when data movement comes into play (1): since there are no permanent addresses stored in the OIDs, the objects can be moved freely. Additionally, object replication and access to different versions can be supported effectively. The price for this flexibility is that the OID has to be mapped in a lookup step to the physical location of (a replica of) the object. The object location table presented in this paper uses logical OIDs.

The aim of our OLT is to enable and optimise access to physics objects which may be replicated. The OLT not only provides lookup operations, but also determines which replica to return in any lookup operation in order to minimise the access cost.

Our OLT is a generic system which can be used for all systems with replication: not only in systems with wide-area replicated data, but also for mass storage systems which stage (replicate) from tape to disk, and for systems that replicate objects while reclustering them, like (2; 3).

1.2 Object location table implementations

There are many ways to implement an OLT that resolves logical OIDs into physical locations. A good introduction to some alternative techniques can be found in (1). Perhaps the most obvious choice for implementing an OLT is to use a hash table or binary tree. (1) shows however that an implementation with *direct mapping* is usually faster. Direct mapping exploits the fact that the logical OID value can be 'invented' by the system when the object is created. In direct mapping, the logical OID is created to encode the exact physical location of a pointer, which in turn specifies the physical location of the object. Resolving the OID involves retrieving the pointer value, which can be done by reading a single disk page only. Thus, in the worst performance case of direct mapping, every object lookup generates a single random disk read operation.

Say that a CMS physicist wants to run an offline physics analysis job, visiting events at ten times the production rate of the experiment. CMS produces 100 events per second. Say the job only accesses one object per event. This translates into $10 * 100 = 1000$ object lookup operations per second. If the OLT is implemented with direct mapping, this implies 1000 random disk read operations per second. A typical disk drive can process only some 100 random

read operations per second (4). So the physicist needs to use 10 disk drives in parallel just for the object lookups, without even retrieving the objects themselves! Obviously, direct mapping, or any other logical OID implementation that can be found in a database system, is too expensive to use, unless something is done to ensure that the worst performance case will not occur much in practice.

Our OLT system exploits a useful property of physics analysis to greatly improve on the worst case above. In physics analysis, at least for CMS, the order in which the individual events are processed is irrelevant. Furthermore, it is preferred to process events in time order, in the order in which they occurred in the detector, because this optimises the access to time-dependent calibration data. We transform this preference into a requirement: our OLT system requires that, at least on the small scale, events are iterated over in time order. This implies that individual objects are always accessed, and thus looked up, in the time order of their corresponding events. By storing location pointer values in the same time order, we can process many concurrent location lookup operations for every single disk page read.

We also get scalability by using a fixed partitioning of the whole event set into *chunks*. A chunk is a set of some 10^4 to 10^6 events, typically a run, a part of a run, or a single physics channel of a (part of a) run. The exact chunk size is not important for our OLT, we expect that the choice of a chunk size will be led by tape performance considerations.

2 The logical OID and the job model

In our system, a logical OID is a unique identifier of a particular object that holds physics data for a particular event. The OID has three fields, as follows.

(*chunk ID*, *event sequence number*, *type/version ID*)

The first two fields, chunk ID and event sequence number, uniquely identify the event to which the object belongs. The event sequence number is simply the position of the event in the chunk to which it belongs. The last field, type/version ID, uniquely identifies the object among the different objects can exist in the system for that event. The type/version ID of a raw data object will uniquely identify the sub-detector that produced the data in the object, and maybe the a version number if the configuration of the subdetector can change. The type/version ID of a reconstructed physics object will uniquely identify the reconstruction algorithm that computed the object, and also the version of that algorithm and of any other versioned data which the algorithm used. Our data model implies that reconstructed objects are versioned: existing

objects are never updated in place.

For the purpose of using our OLT system, a physics job is defined as follows. Every job specifies:

- (1) An event set S of the events which must be visited. We assume that each event in S is represented by its (*chunk ID*, *event sequence number*) pair.
- (2) A *job function*, which is a piece of executable C++ code (typically a loadable shared library), to be run for each event.
- (3) For every different kind of object to be read for each event by the job, the type/version ID of that object.
- (4) (Optional) If, for some type/version ID t , the corresponding object is not needed for every event in S , then a smaller event set $S_t \subset S$ can be specified, with the understanding that the job function will only need the objects t for the events in S_t .

To execute the job, the system first computes the set C of all chunks that contain one or more objects identified in S . For every chunk in C , the system runs a *sub-job* over the requested events in this chunk. This iteration is always done in event time order, so in the order of increasing event sequence number.

3 The Object Location Table

Figure 1 depicts the physical representation of the whole object location table in the two top quadrants. In the two bottom quadrants the physical objects and their distribution to different locations is shown. In this example, three different *collections* (a set of replicated objects is called a collection) exist at two geographically distributed sites named CERN and RC1. The site CERN holds two collections, one on tape and the other one on disk. While the collection on tape contains all possible objects of the corresponding chunk and type, the collection on disk only stores a subset of the objects. Another subset is stored in the collection on disk in RC1.

We assume a distributed system with interfaces to the data store at CERN and RC1. Hence, a local copy (replica) of the object location table is available at both sites (see top quadrants). We can distinguish between five different index levels where the first four ones are replicated to all sites where objects can be looked up and the lowest level is only stored where the physical objects are located.

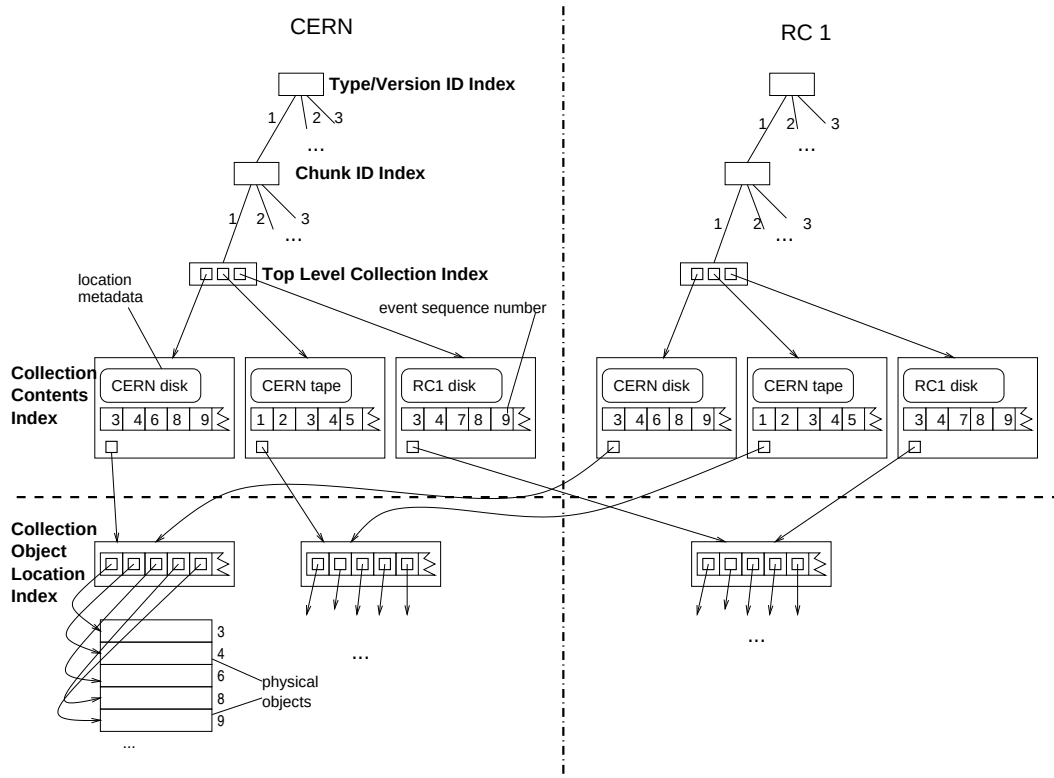


Fig. 1. Physical representation of the whole object location table, with different levels of indexes. Arrows indicate location dependent, physical pointers.

4 Use of the Object Location table in sub-job execution

Our OLT automatically determines which replicated objects to access in order to minimise the access cost. This optimisation process happens at the level of the execution of a single sub-job.

We show the use and functioning of our OLT by taking a single sub-job as a running example. We take a sub-job, of some physics analysis job in the model of section 2, that should access the objects with the type/version ID 80 for four events in a set S_5 in chunk 5. (A realistic sub-job would access objects for much more events.) As the chunk ID and type/version ID are fixed for the whole sub-job, we can represent the identity of the objects needed using the event sequence numbers in their logical OIDs only. Say we need the set

$$O = \{2, 4, 5, 8\}$$

Say that traversing the OLT with the type/version ID 80 and chunk ID 5 finds four collections that are relevant for the job. The contents of these collections can again be represented as sets of event sequence numbers.

collection 1 = $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ located at CERN on tape

collection 2 = { 2, 5, 8 } located at CERN on disk
collection 3 = { 2, 4, 8, 10 } located at CERN on disk
collection 4 = { 2, 4, 8, 10 } located at RC1 on disk

Say the (sub-)job runs at RC1. We can now ask the question: which replicas should be read by the sub-job to obtain the best overall performance? Looking at the information above, in this case the answer is obvious:

Read object 2 from collection 4
Read object 4 from collection 4
Read object 5 from collection 2
Read object 8 from collection 4

For actual, much larger, sub-jobs, the above list is very long. However, the optimal way of reading objects can be represented in a much more compact way. We call this compact representation a *schedule*. The schedule defines exactly from which collection any needed object should be read. In our approach, a schedule is represented by an ordered list of collections. For the example above, the corresponding schedule is:

Collection 4, Collection 2.

The schedule should be used as follows. To decide from which collection any given object should be read, look at the collections in the schedule from left to right. The first collection that contains the object is the one from which it should be read.

By introducing the idea of a schedule, we have split the problem of using the OLT to optimise access into two phases:

- (1) Compute the optimal schedule for the sub-job
- (2) Run the sub-job and use this schedule to do the actual lookups and reading

Below we cover the use of the schedule first.

4.1 Use of the schedule

We propose the following architecture for the OLT mechanism that runs during a sub-job. Large parts of this architecture have been implemented and studied as part of earlier work (3).

At the start of the sub-job, the OLT creates an *object iterator* for every separate type of object requested by the sub-job. In our running example there will

be one such iterator i corresponding to the type/version ID 80. This iterator is handed to the physics code (the job function) that runs inside the sub-job. A `next()` method can be called to obtain the next object in the requested set (the set O), these objects are delivered in the order of increasing increasing event sequence number. At the start of the sub-job, the OLT implementation also creates a *collection iterator* c_i for every collection in the in the schedule. These collection iterators are internally bound to the object iterator i : whenever $i.next()$ is called, i will query the collection iterators c_i , in the order specified by the schedule, until it has one that delivers (a pointer to a replica of) the currently needed object. The collection iterators c_i thus have a method `try_to_get_object(o)` which can return either a 'not present' code, or a pointer to the o that is present in the collection.

For an iterator c_i over a collection on local disk (as was implemented in (3)), the implementation can suffice with advancing a lookup pointer over the local Collection Contents Index of the collection in question: this pointer can be advanced in lock-step with the `try_to_get_object` requests coming in. When an object is found to be present in the collection, the corresponding pointer in the Collection Object Location Index can be used.

As a more complex example of the implementation of a collection iterator c_i , consider an iterator for a remote collection. Suppose that the most efficient way to access the set O_i of needed objects from this collection is to package the objects into a file first, and then download the file with a fast FTP mechanism to create local copies of the objects. If this is the case, then c_i will first compute O_i using the schedule, the set O , and the index level data in the OLT. Then, O_i is handed to a remote mechanism that packages the needed objects, adds an index to the package (similar to a Collection Object Location Index), and transfers it to the local site. With this implementation, the sub-job will only start iterating when the package transfer is complete. Of course, a more complex implementation of c_i that can create an overlap in the operations is also feasible.

4.2 Computing the optimal schedule

To introduce the problem of computing the optimal schedule we first consider the possible list of schedules SL . In the previous example we have four collections. Some possible schedules are where the numbers represent collection numbers

$$SL = \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \dots, \underline{\{2, 4\}}, \dots\}$$

The optimum schedule in the running example is $\{2,4\}$. Finding the optimum means choosing a single schedule s from SL so that

- the schedule is feasible: all the objects which are requested by the sub-job have to be contained in the collections in the schedule
- the cost of the schedule, according to some cost function, is minimal.

We define the cost of a schedule as the sum of the *usage costs* for all collections in the schedule. The *usage cost* for a collection is the value of a collection-specific cost function defined by the implementer. In computing the usage cost, several factors can be considered, like the

- the number of objects needed from the collection according to this schedule
- the size of the collection in bytes
- the load of the data server,

and if the collection is remote, there can be additional factors like

- the cost of the method by which the objects are accessed and transferred
- the network load,
- costs that reflect policies governing remote access

The above represents a very general framework, and finding appropriate cost functions for collections (depending on their location) is not trivial, especially if policy issues have to be encoded within them. However, based on earlier prototyping experience (2; 3), we argue that the cost functions need not be absolutely perfect: reasonable cost functions will generally yield reasonable results. Furthermore, our prototyping experience indicates that the cost of using the above general framework is not prohibitive. Good schedules can be computed in a reasonable time.

4.3 Operational experience in computing schedules

Our implementations for computing schedules in (2) and (3) use a two-phase approach. In phase 1, statistics are gathered about how the collections relate to each other and to the set of requested objects. A typical item in these statistics (using our running example) would be:

There is a class of objects which are in all collections 1, 3, and 4, but not in 2. The size of this class is 3 objects. 2 of these objects are requested by the sub-job.

All needed statistics can be made very quickly, in the time needed to read all indices and the set O from disk, by using a set-algorithm that exploits the

fact that all sets concerned are represented as sorted lists of integers.

In phase 2 of this approach, the solution space SL of schedules is searched using a branch-and-bound algorithm. To compute the cost of a candidate schedule, the number of objects needed from each collection in that schedule needs to be computed. This can be done very quickly using the statistics gathered in phase 1. These numbers are then fed into the collection-level cost functions.

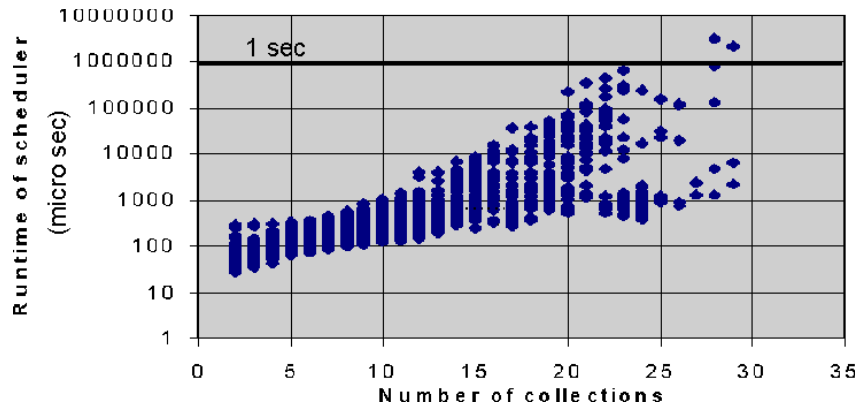


Fig. 2. Time complexity of phase 2 of our implemented scheduler, finding the optimal schedule

Figure 2 shows the time complexity of phase 2 of our implemented scheduler, running inside the simulated multi-user physics analysis system described in (2). In this system, the number of different collections per unique chunk ID and type/version ID combination varies from 1 up to some 28. The optimal schedule is generally found in less than 1/10th of a second. Although the scheduling problem is NP-complete in theory (3), in practice good solutions can be found quickly for interesting problem sizes. Note that Figure 2 shows the time complexity for finding the optimal schedule: if the requirement for an optimum is dropped, heuristics (5) can be used to quickly find a good solution for much larger problem sizes. For our purposes, the above two-phase scheduling method was fast enough, so faster heuristics were not explored. It is clear though that, if more speed or scalability is needed, the opportunities for improvement over our approach are far from exhausted.

5 Conclusions

The use of replication implies the use of a table to resolve the logical, location independent object descriptor into a physical location where an object replica can be found. For modern physics experiments the table needs to scale to at least some 10^{10} objects. Our OLT obtains its scalability by limiting the randomness of the lookup operations that are performed on the table. There is a 'locality of access' on the lookup table, and this ensures that we can

process many subsequent object lookup operations for every single index page read from disk.

Apart from scalable lookup operations, the OLT also supports scalable update operations. In our update model, whole collections are added to, or deleted from, the OLT. For example, in figure 1, to replicate some more objects to RC1, a new collection with these objects is created at that location, and appropriate Collection Contents Indices are added to the OLT when this is done. As a result, two collections will now exist at the 'RC1 disk' storage level. The OLT does not require a costly step in which these two collections or their indexes are merged: it supports transparent access to the objects in either of the collections.

Large parts of the OLT system have been implemented (2; 3). However, with respect to finding good collection-level cost functions, especially if these have to reflect policies governing remote access, much work remains to be done.

References

- [1] A. Eickler, A. Gerlhof, D. Kossmann. A Performance Evaluation of OID Mapping Techniques, Proc. of the Conf. on Very Large Data Bases (VLDB), Zurich, Switzerland, September 1995.
- [2] K. Holtman, P. van der Stok, I. Willers. A Cache Filtering Optimisation for Queries to Massive Datasets on Tertiary Storage. Proc. of DOLAP'99, Kansas City, USA, November 6, 1999.
- [3] K. Holtman, P. van der Stok, I. Willers. Automatic Reclustering of Objects in Very Large Databases for High Energy Physics, Proc. of IDEAS '98, Cardiff, UK, p. 132-140, IEEE 1998.
- [4] K. Holtman, J. Bunn, Scalability to Hundreds of Clients in HEP Object Databases, Proc. of CHEP '98, Chicago, USA 1998.
- [5] T. Grossman, A. Wool. Computational experience with approximation algorithms for the set covering problem. Euro. J. Operational Research, 101(1):81-92, August 1997.