Optimizing Performance of Schema Evolution Sequences *

Kajal T. Claypool, Chandrakant Natarajan, and Elke A. Rundensteiner

Worcester Polytechnic Institute, 100 Intitute Road,
Worcester, MA, USA
{kajal, chandu, rundenst}@cs.wpi.edu
http://davis.wpi.edu/dsrg

Abstract. More than ever before schema transformation is a prevalent problem that needs to be addressed to accomplish for example the migration of legacy systems to the newer OODB systems, the generation of structured web pages from data in database systems, or the integration of systems with different native data models. Such schema transformations are typically composed of a sequence of schema evolution operations. The execution of such sequences can be very time-intensive, possibly requiring many hours or even days and thus effectively making the database unavailable for unacceptable time spans. While researchers have looked at the deferred execution approach for schema evolution in an effort to improve availability of the system, to the best of our knowledge ours is the first effort to provide a direct optimization strategy for a sequence of changes. In this paper, we propose heuristics for the iterative elimination and cancellation of schema evolution primitives as well as for the merging of database modifications of primitives such that they can be performed in *one* efficient transformation pass over the database. In addition we show the correctness of our optimization approach, thus guaranteeing that the initial input and the optimized output schema evolution sequence produce the same final schema and data state. We provide proof of the algorithm's optimality by establishing the confluence property of our problem search space, i.e., we show that the iterative application of our heuristics always terminates and converges to a unique minimal sequence. Moreover, we have conducted experimental studies that demonstrate the performance gains achieved by our proposed optimization technique over previous solutions.

Keywords: Schema Evolution, Object-Oriented Databases, Optimization.

* This work was supported in part by the NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and Informix for software contribution. Kajal T. Claypool would like to thank GE for the GE Corporate Fellowship. Special thanks also goes to the PSE Team specifically, Gordon Landis, Sam Haradhvala, Pat O'Brien and Breman Thuraising at Object Design Inc. for not only software contributions but also for providing us with a customized patch of the PSE Pro2.0 system that exposed schema-related APIs needed to develop our tool.

K.R. Dittrich et al. (Eds.): Objects and Databases 2000, LNCS 1944, pp. 114–127, 2001.

1 Introduction

Not only is it difficult to pre-determine the database schema for many complex applications during the first pass, but worst yet application requirements typically change over time. For example [17] documents the extent of schema evolution during the development and the initial use of a health management system at several hospitals. There was an increase of 139% in the number of relations and an increase of 274% in the number of attributes, and every relation in the schema was changed at least once during the nineteen-month period of the study. In another study [13], significant changes (about 59% of attributes on the average) were reported for seven applications which ranged from project tracking, real estate inventory and accounting to government administration of the skill trades and apprenticeship programs. These studies reveal that schema changes are an inevitable task not only during the development of a project but also once a project has become operational. For this reason, most object-oriented database systems (OODB) today support some form of schema evolution [18,3, 14,10].

The state of the art in OODB evolution is to offer seamless change management by providing transparency between the database schema and the application source files representing the schema. Systems such as ObjectStore [14] allow users to directly edit the application source leaving the OODB system to handle the propagation of this sequence of changes to the database schema. Other systems such as Itasca [10] provide a graphical user interface (GUI) that allows the users to specify a set of their schema changes graphically while the system again is responsible for propagating the schema changes to the database. Other systems such as O_2 [18,2], TESS [12] and SERF [5] all deal with more advanced schema changes, such as merging of two classes, which are often composed of a sequence of schema evolution primitives. All of these systems deal with applying not a *single schema change* but a *sequence of schema changes* to the underlying database.

Unfortunately, schema evolution remains a very expensive process both in terms of system resource consumption as well as database unavailability [9]. Even a single simple schema evolution primitive (such as add-attribute to a class) applied to a small database of 20,000 objects (approx. 4MB of data) has been reported to already take about 7.4 minutes [7]. With the current database technology for the specification of schema changes as a sequence of changes the performance of the system is further compromised, with evolution costs for large databases possibly escalating to hours, perhaps even days for entire sequences.

In previous work, researchers have looked at improving system availability during schema evolution by proposing execution strategies such as deferred execution [18,9]. No work, however, has been undertaken to actually optimize or reduce the sequence of operations that are being applied to a given schema. Kahler et al. [11] have looked at pre-execution optimization for reducing the number of update messages that are sent to maintain replicated sites in the context of distributed databases. In their approach, the messages are simple data updates on tuples. He sorts the number of messages by their tuple-identifier, and

then condenses (with merge or remove) the change history of the tuple into one update operation.

We now present a similar approach (merge, cancel, eliminate) for optimizing a sequence of schema evolution operations. Our optimization strategy, called CHOP, exploits two principles of schema evolution execution within one integrated solution:

- Minimize the number of schema evolution operations in a sequence by for example canceling or eliminating schema evolution operations. For example, adding an attribute and then deleting the same attribute is an obvious case of cancellation where neither operation needs to be executed.
- Merge the execution of all schema evolution changes that operate on one extent to amortize the cost of schema evolution over several schema changes. For example, consider a sequence that adds two or more attributes to the same class. Object updates for these done simultaneously can potentially half the cost of executing these sequentially.

To the best of our knowledge ours is the first effort to provide an optimization strategy for a sequence of changes prior to execution a la Kahler [11]. Our approach is orthogonal to the existing execution strategies for schema evolution, i.e., it can in fact be applied to both immediate and the deferred execution strategies [9].

To summarize, in this paper, we present a general strategy for the reduction of a given sequence of schema evolution operations **prior** to its actual execution. Our work is based on a taxonomy of schema evolution operations we developed for the ODMG object model but it can easily be applied to any other object model. We present here an analysis of the schema evolution operations and of the schema to characterize the conditions under which operations in a sequence can be optimized. Based on this analysis we present the *merge*, *cancel* and *eliminate* optimization functions and the conditions under which they can be applied. We have also been able to show both formally and experimentally that the order in which these functions are applied is not relevant for the final optimized sequence, i.e., they will all produce the same *unique* final sequence. As a conclusion to our work we also present a summary of our experimental results.

Outlook. The rest of the paper is organized as follows. Section 3 presents the taxonomy of schema evolution primitives on which we base our analysis. Section 4 gives a formalization of the schema evolution operation properties. Section 5 presents the actual optimization functions while Section 6 describes how these are combined to form the overall CHOP strategy. Section 7 presents our experimental evaluation. We conclude in Section 8.

2 Related Work

Current commercial OODBs such as Itasca [10], GemStone [3], ObjectStore [14], and O_2 [18] all provide some schema evolution - be it a set of schema evolution

primitives or some mechanism for global changes. Some work in schema evolution has focused on the areas of optimization and database availability, in particular on deciding when and how to modify the database to address concerns such as efficiency and availability. In [9,8], a deferred execution strategy is proposed for the O_2 database system that maintains a history of schema evolution operations for a class and migrates objects only when actually accessed by the user. This allows not only for high database availability but also amortizes the cost of the object transformations with that of a query lookup. However, no optimizations are applied to this sequence of schema evolution operation(s) and the performance of this deferred mechanism deteriorates as the set of queried objects grows larger. Our approach, while primarily optimizing the immediate update mode, also complements the deferred mechanism by offering time savings as the queried set of objects and the number of schema evolution operations to be applied on it grows larger.

Similar work has also been done in database log recovery mechanisms. In [11], Kahler et. al use hashing to compute the net effect of changes in a log by eliminating and/or collapsing redundant log enteries due to insert-remove pairs, replace-remove pairs, etc. We use a similar approach (cancel and eliminate optimizations) for removing redundancy from a sequence of schema evolution sequences. The focus of work is however, beyond simply condensing all changes into one change. In this work, we have shown that if one or more optimizations are applied, giving precedence to one over the other is immaterial as they all lead to the same minimal optimized sequence of operations. Segev et al. have applied similar pre-execution optimizations for updating distributed materialized views since data sequences of updates must be shipped from the base tables to the view [16].

3 Background - Taxonomy of Schema Evolution Primitives

The CHOP approach is based on the Object Model as presented in the ODMG Standard [4]. The ODMG object model encompasses the most commonly used object models and standardizes their features into one object model, thus increasing the portability and applicability of our prototype. Here we present a taxonomy of schema evolution primitives (Table 1) that we have defined for the ODMG model based on the other data models as O_2 or Itasca [18,3,14,10]. The taxonomy given in Table 1 is a **complete** set of schema evolution primitives such that it subsumes every possible type of schema change [1]. The taxonomy is the **essential** [19] set of schema evolution primitives, i.e., each primitive updates the schema and the objects in a unique way that cannot be achieved by a simple composition of the other primitives.

Table 1. The Taxonomy of Schema Evolution Primitives.

Term	Description	Capacity Effects
add-class(c, $\mathcal{C}$)	Add new class c to $\mathcal{C}$ in schema S (AC)	augmenting
delete-class(c)	Delete class c from $\mathcal{C}$ in schema S if *subclasses(C)* $= \emptyset$ (DC)	reducing
rename-class(c_x, c_y)	Rename class c_x to c_y (CCN)	preserving
add-ISA-edge(c_x, c_y)	Add an inheritance edge from c_x to c_y (AE)	augmenting
delete-ISA-edge(c_x, c_y)	Delete the inheritance edge from c_x to c_y (DE)	reducing
add-attribute(c_x, a_x)	Add attribute a_x to class c_x (AA)	augmenting
delete-attribute(c_x, a_x)	Delete the attribute a_x from the class c_x (DA)	reducing
rename-attribute(a_x, b_x, c_x)	Rename the attribute a_x to b_x in the class c_x (CAN)	preserving

4 Foundations of Schema Evolution Sequence Analysis

To establish a foundation for our optimization principles we have developed a formal characterization of the schema evolution operations, their impact on the schema, as well as their interactions within a sequence. Due to space constraints we only summarize these characterizations here. For more details please see [6].

Table 2 defines the various relationships that can exist in general between schema evolution operations. Table 3 applies these to the schema evolution operation presented in Table 1.

Table 2. Classification of Operation Properties.

Operation Relation	**Description**
same-operation-as	`op1` is **same-operation-as** `op2` if they both have the same operation name irrespective of the particular parameters they are being applied to.
inverse-operation-of	`op1` is **inverse-operation-of** `op2` if the effects of one operation `op1` could be canceled (reversed) by the effects of the other operation `op2`.
super-operation-of	`op1` is **super-operation-of** `op2` if the functionality of `op1` superimposes the functionality of `op2`, i.e., `op1` achieves as part of its functionality also the effects of `op2`

Table 3. Classification of Operation Properties for the Schema Evolution Taxonomy in Table 1 (with **same** = **same-operation-as**, **inverse** = **inverse-operation-of** and **super** = **super-operation-of**).

	AC()	**DC()**	**CCN()**	**AA()**	**DA()**	**CAN()**	**AE()**	**DE()**
AC()	same	inverse	-	super	-	-	-	-
DC()	inverse	same	super	super	super	super	super	super
CCN()	-	-	same/inverse	-	-	-	-	-
AA()	-	-	-	same	inverse	-	-	-
DA()	-	-	-	inverse	same	super	-	-
CAN()	-	-	-	-	-	same	-	-
AE()	-	-	-	-	-	-	same	inverse
DE()	-	-	-	-	-	-	inverse	same

However, for optimization it is not sufficient to categorize the schema evolution operations based on just their functionality. It is important to also know the parameters, i.e., the context in which these operations are applied. Table 4 presents the schema element relationships.

Table 4. Classification of Schema Element Relations.

Schema Relation	**Description**
definedIn	gives the scope for all schema elements (from ODMG).
extendedBy	gives the inheritance relationship of schema elements of the type `Class` (from ODMG).
same-as	gives the identity of a class or a property based on unique name in given scope (CHOP extension).
aliasedTo	gives the derivation of a schema element from another element through a series of name modifications (CHOP extension).

While the *operation properties* (Table 2) and the *context properties* (Table 4) provide *necessary* criteria for when an optimization function can be applied, they are not always sufficient in the context of a sequence. Here we briefly summarize the relationships of operations in a sequence.

- **Schema-Invariant Order Property.** When operation `op1` is **sameAs** `op2`, we identify the **schema-invariant-order** property as:
  - For two **capacity-augmenting** and **capacity-reducing** operations, `op1` is in **schema-invariant-order** with `op2` if the order of their parameters is the **same**.
  - For **capacity-preserving** operations, `op1` is in **schema-invariant-order** with `op2` if the order of their parameters is **reversed**.

- **Object-Invariant-Order Property -** `op1` is **object-invariant-order** with `op2`, if `op1` is **capacity-augmenting** and `op2` is **capacity-reducing** and in the sequence of evolution operations the **capacity-augmenting** operation appears prior to the **capacity-reducing** operation. There is no specific **object-invariant-order** for the **capacity-preserving** operations.
- **Dependency Property -** The schema elements used as parameters by the two operations `op1` and `op2` being considered for optimization must not be referred to by any other operation which is placed between the two operations in the sequence.

5 The CHOP Optimization Functions

The integral component of the CHOP optimization algorithm are the optimization functions that can be applied to pairs of schema evolution operations within the context of their resident schema evolution operation sequence. In this section we present the general description of an optimization function and three instantiations of the optimization functions (*merge*, *cancel*, *eliminate*), that we have formulated for the optimization of the primitive set of schema evolution operations given in Section 3.

5.1 An Optimization Function

The crux of the CHOP optimization algorithm is an optimization function which takes as input two schema evolution operations and produces as output zero or one schema evolution operation, thereby reducing the sequence of the schema evolution operations. Formally, we define an optimization function as follows:

Definition 1. *Given a schema evolution operation sequence* Σ, `op1`, `op2` ... `opn` *with* `op1` *before* `op2` *(i.e., if the index position* `i` *of* `op1` *is less than the index position* `j` *of* `op2` *in* Σ, *index(op1)* $= i < j =$ *index(op2)), an optimization function* F_Σ *produces as output an operation* `op3` *which is placed in the sequence* Σ *at the index position* `i` *of the first operation,* `op1`. *The operation at index position* `j` *is set to a* `no-op`. *The operations* `op1`, `op2` *and* `op3` *can be either schema evolution primitives as described in Table 1 or complex evolution operations as defined in Section 5.2.*

A major requirement for the CHOP optimization is to reduce the number of schema evolution operations in a sequence such that the final schema produced by this optimized sequence is consistent and is the same as the one that would have been produced by the unoptimized sequence for the same input schema. Thus, an optimization function must not in any way change the nature of the schema evolution operations or the order in which they are executed. Towards that goal, any optimization function must observe several properties characterized below.

Invariant-Preserving-Output Operations. Schema evolution operations guarantee the consistency of the schema and the database by preserving the invariants defined for the underlying object model [1]. An important property of the optimization function therefore is for its output (any of its output operations) to also preserve the schema consistency by preserving the invariants defined for the object model.

Schema-State Equivalent. Above all an optimization function must guarantee correctness, i.e., the schema produced by output of the optimization function (optimized sequence) must be the same as the schema produced by the unoptimized sequence when applied to the same input schema. This property can in fact be proven formally. In [6] we provide a formal set of definitions and proofs for this.

Relative-Order Preserving. As discussed in Section 4, the order in which the schema evolution operations appear with respect to one another in a sequence is relevant to the application of an optimization function. This **relative order** of an operation `op1` in a sequence is defined by its index, `index(op1)`, with respect to the index of the other operations, e.g., `index(op2)`, in the sequence. For example, if `index(op1)` $<$ `index(op2)`, then `op1` is *before* `op2` in the sequence, denoted by `op1` $<$ `op2`. Operations executed out-of-order can cause unexpected variance in the final output schema. For example, consider two operations in a sequence with the order as given here: $<$ `DA(C,a)`, `AA(C,a)` $>$. When executed in the order given the attribute `a` is first deleted from the class `C` and then re-added. However, all of the information stored in the attribute `a` is lost. Now, switching the order of execution of the two operations leads to a very different schema.

Thus, it is essential for an optimized sequence to preserve the **relative-order** of the input sequence.
Using the above stated properties, we now refine the definition of an optimization function as follows:

Definition 2 (Optimization Function.). *Any optimization function in CHOP defined as in Definition 1 must be* **invariant-preserving**, **schema-state-equivalence preserving** *and* **relative-order preserving**.

For the CHOP approach, we define three such optimization functions, Merge, Eliminate and Cancel.

5.2 The Merge Optimization Function

The time taken for performing a schema evolution operation is largely determined by the page fetch and page flush times [7]. In our proposed CHOP approach we amortize the page fetch and flush costs over several operations by collecting

all transformations on the same set of objects and performing them simultaneously[1].

A collection of schema evolution operations for the same class which affect the same set of objects, i.e., it is possible to perform all the object transformations for these operations during the same page fetch and flush cycle, is called a **complex operation** denoted by $<$ op$_1$, ... op$_k$ $>$, with k $\geq$ 2. For two complex operations, op1 $=<$ op$_i$.... op$_j$ $>$ and op2 $=<$ op$_m$.... op$_n$ $>$, the operation pairs (op$_j$,op$_m$) and (op$_n$, op$_i$) are termed **complex-representative pairs**.

Definition 3. Merge *is an optimization function (Definition 2) that takes as input a pair of schema evolution operations, either primitive or complex,* op1 *and* op2*, and produces as output a complex operation* op3 $=<$ op1, op2 $>$*. If one or both of the input operations are a complex operation, e.g.,* op1 $=<$op$_i$, op$_j$ $>$ *and* op2 $=<$op$_m$, op$_n$ $>$*, then a relative order within the complex operations* op$_3$ *is maintained such that the output operation* op3 = $<$op$_i$, op$_j$, op$_m$, op$_n$ $>$*. The input operations* op1 *and* op2 *must satisfy:*

- **Context Property**
  - *If* op1 *and* op2 *are related by the* **same-operation-as** *property, then their context parameters must be* **definedIn** *in same scope.*
  - *If* op1 *and* op2 *are related by the* **super-operation-of** *property, then for the sub-operation the* **definedIn** *scope of the context must be the* **sameAs** *the context of the super-operation.*
  - *If* op1 *and* op2 *are related by the* **inverse-operation-of** *property, then the context of* op1 *must be* **sameAs** *the context of* op2 *and* **definedIn** *same scope.*
- **Dependency Property** *must hold.*

When one or both of the input operations op1 *and* op2 *are complex, then all the merge conditions given above must be satisfied by at least one pair of operations in the complex operation. This is the* **complex-representative pair**.

For example, given an operation that adds a `name` attribute to a class called `Employee` and a subsequent operation that adds a `age` attribute to the same class `Employee`, we can merge the two operations they are related by the **same-operation-as** property and their context parameters are **definedIn** the same scope, i.e., `Employee` for both operations is in the same schema and the attributes `name` and `age` are being added to the same class `Employee`. Lastly, the **dependency** property holds as there are no operations between the index positions of the two operations.

A complex operation is thus a sub-sequence of schema evolution operations and other optimization functions (cancel and eliminate) can be applied on the primitive schema evolution operations inside of a complex operation. However, the merge optimization function itself cannot be applied inside of a complex operation as this can lead to infinite recursion.

[1] This merge of operations relies on the underlying OODB system to be able to separate the schema evolution operation into a schema change at the Schema Repository level and into object transformations at the database level.

5.3 The Eliminate Optimization Function

In some cases a further optimization beyond merge may be possible. For example, while it is possible to merge `DA(Employee, name)` and `DC(Employee)` the execution of `DC(Employee)` makes the prior execution of `DA(Employee, name)` redundant. Hence, some operations may be optimized beyond a merge by being completely **eliminated** by other operations, thus reducing the transformation cost by one operation.

Definition 4. Eliminate *is an optimization function as defined in Definition 2 that takes as input a pair of schema evolution primitives* `op1` *and* `op2` *and produces as output* `op3`*, such that* `op3` = `op1` *if* `op1` = **super-operation-of** *(*`op2`*) or* `op3` = `op2` *if* `op2` = **super-operation-of** *(*`op1`*). The input operations* `op1` *and* `op2` *must satisfy:*

- **Operation Property** *such that either* `op1` = **super-operation-of** *(*`op2`*) or* `op2` = **super-operation-of** *(*`op1`*),*
- **Context Property** *such that the* **definedIn** *scope of the sub-operation is* **sameAs** *the context parameter of the super-operation, and*
- **Dependency Property** *must hold.*

5.4 The Cancel Optimization Function

In some scenarios further optimization beyond a merge and eliminate may be possible. Some schema evolution operations are inverses of each other, for example, `AA(Employee, age)` adds an attribute and `DA(Employee, age)` removes that attribute. A **cancel** optimization thus takes as input two schema evolution operations and produces as output a no-op operation, i.e., an empty operation that does nothing.

Definition 5. Cancel *is an optimization function as in Definition 2 which takes as input a pair of schema evolution primitives* `op1` *and* `op2` *and produces as output* `op3`*, where* `op3` = `no-op`*, an empty operation, assuming the input operations* `op1` *and* `op2` *satisfy:*

- **Operation Property** *such that* `op1` *and* `op2` *are related by the* **inverse-operation-of** *property,*
- **Context Property** *such that* `op1` *and* `op2` *are* **definedIn** *the same scope and* `op1` *is* **sameAs** `op2`*,*
- **Schema-Invariant-Order Property** *for capacity-reducing operations must hold, and*
- **Object-Invariant-Order Property** *must hold, and*
- **Dependency Property** *must hold.*

6 CHOP Optimization Strategy

The CHOP optimization algorithm iteratively applies the three classes of optimization functions **merge**, **eliminate** and **cancel** introduced in Section 5 until the algorithm terminates and a minimal solution is found.

However, before we can address the issue of minimality it is necessary to examine two issues: (1) if one or more functions are applicable, is choosing the right function essential? and (2) when there are more than one pair of operations that can be optimized, is choosing the right pair essential?

Choosing the Right Optimization Function. We note that the conditions under which the **merge** optimization function can be applied is a *superset* of the conditions under which an **eliminate** or a **cancel** can be applied, while the conditions under which a **cancel** and an **eliminate** can be applied are mutually exclusive. Thus, often a **merge** can be applied to a pair of operations where either an **eliminate** or a **cancel** can be also applied. However, as these optimizations offer different degrees of reduction for a pair of schema evolution operations (with **merge** offering the least and **cancel** the most), choosing the optimization function that offers the most reduction is very desirable.

We can however formally show that doing a **merge** where a **cancel** or an **eliminate** is also applicable does not prevent the application of a **cancel** or an **eliminate** during the next iterative application of these functions. A formalization of this property and its proof can be found in [6].

Operation Dependencies and Optimization Functions. An important criteria for the successful application of any of the three optimization functions is that the *Dependency Property* as given in Section 4 must hold. That is, there must be no reference to the schema elements used as parameters in the two operations op_1 and op_2 being considered for optimization by any other operation which is placed between the two operations in the sequence. However, the order in which the pairs of operations are selected can have an effect on this dependency.

Consider a sequence of three operations op_1, op_2 and op_3. Consider that the pairs (op_1, op_2) and (op_2, op_3) can be immediately optimized while a successful optimization of the pair (op_1,op_3) requires removing the dependency operation op_2. In this case, there are two possibilities for applying the optimization functions on the pairs of operations. We could either apply the respective optimization functions on the pair (op_1, op_2) and then on the pair (op_2, op_3) [2] and not be concerned about the optimization possibility between $op_1$and op_3. Or we could first apply the optimization function on the pair (op_2, op_3), reduce the dependency $op_2$and then optimize the pair (op_1, op_3). However, as before our goal is to achieve the *maximum* optimization possible.

[2] Note that in some cases op_2may not exist any more and hence optimizing (op_2, op_3) may no longer be possible.

We can formally show that the order of selection for pairs of schema evolution operations in a sequence for the application of one of the three optimization functions does not prevent the achievement of *maximum* optimization [6].

Confluence. While the main goal for the optimization is to achieve maximum optimization possible in an effort to reduce schema evolution costs, we also want to keep the overhead of optimizing to a minimal. However, there are multiple permutations and combinations of the optimization functions and the pairs of schema evolution operations that can potentially achieve the maximum optimization. Enumerating all the possible choices prior to selecting one for execution results in an exponential search space. This overhead from enumerating these choices alone would cancel any potential savings achieved by the optimization.

However, based on the function properties in [6], we can show that all possible combinations of optimization functions for a given sequence converge to one **unique minimal**, thereby eliminating the need to enumerate all the possible choices. The following states the theorem of confluence. We have formally and experimentally proven this result [6].

Theorem 1. [Confluence Theorem]: *Given an input schema evolution sequence, Σ_{in}, all applicable combinations of optimization functions f_i produce minimal resultant sequences Σ_i that are all exactly the same.*

7 Experimental Validation

We have conducted several experiments to not only evaluate the potential performance gains of the CHOP optimizer. Our experimental system, CHOP, was implemented as a pre-processing layer over the Persistent Storage Engine (PSE Pro2.0). All experiments were conducted on a Pentium II, 400MHz, 128Mb RAM running WindowsNT and Linux. We used a `payroll` schema (refer [6]). The schema was populated with 5000 objects per class in general or are otherwise indicated for each individual experiment. Due to lack of availability of a benchmark of typical sequences of schema evolution operations, the input sequences themselves were randomly generated sequences.

The applicability of CHOP is influenced by two criteria, the performance of the optimized vs the unoptimized sequence of schema evolution operations, and the degree of optimization achievable on average by the optimization functions of CHOP. Due to space limitations we only present a brief summary of our experimental observations. The details can be found in [6].

- The SE processing time for a sequence is directly proportional to the number of objects in the schema. Hence, for larger databases we can potentially have larger savings.
- The optimizer algorithm overhead is negligible when compared to the overall cost of performing the schema evolution operations themselves. Thus our optimization as a pre-processor offers a win-win solution for any system handling sequences of schema changes (Figure 1).

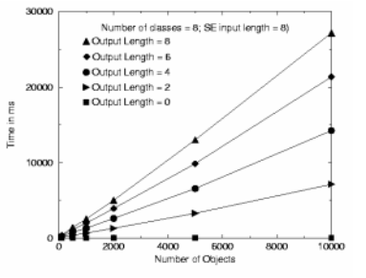

Fig. 1. Best and Worst Case Sequence Times w/o Algorithm Overhead for Input Sequences of Length 8 on the Sample Schema.

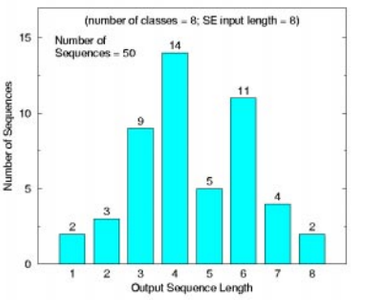

Fig. 2. Distribution: Number of Classes = Sequence Length

- The degree of optimization increases with the increase in the number of class-related operations in the sequence. Hence, depending on the type of sequence, major improvements are possible (Figure 2).
- A random application of the optimization functions on the same sequence resulted in the same final sequence of schema evolution operations.
- We have experimentally tested that on a small-sized database of 20,000 objects per class, even the removal of a single schema evolution operation on a class already results in a time saving of at least 7000 ms. This time savings is directly proportional to the number of attributes and the extent size of a class thus offering huge savings for today's larger and larger database applications.

8 Conclusions

In this paper, we have presented the first optimization strategy for schema evolution sequences. CHOP minimizes a given schema evolution sequence through the iterative elimination and cancellation of schema evolution primitives on the one hand and the merging of the database modifications of primitives on the other hand. Important results of this work are the proof of correctness of the CHOP optimization, a proof for the termination of the iterative application of these functions, and their convergence to a unique and minimal sequence. A version of this system along with the SERF system has been implemented and was presented as a demo at SIGMOD'99 [15]. We have performed experiments on a prototype system that clearly demonstrate the performance gains achievable by this optimization strategy. For random sequences an average optimization of about 68.2% was achieved.

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research.

References

1. J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.
2. P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.
3. R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.
4. Cattell, R.G.G and et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.
5. K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.
6. K.T. Claypool, C. Natarajan, and E.A. Rundensteiner. Optimizing the Performance of Schema Evolution Sequences. Technical Report WPI-CS-TR-99-06, Worcester Polytechnic Institute, February 1999.
7. C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, S. Zaniolo, C. Ceri, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.
8. F. Ferrandina, T. Meyer, and R. Zicari. Correctness of Lazy Database Updates for an Object Database System. In *Proc. of the 6th Int'l Workshop on Persistent Object Systems*, 1994.
9. F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, pages 261–272, 1994.
10. Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.
11. Bo Kähler and Oddvar Risnes. Extending logging for database snapshot refresh. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 389–398. Morgan Kaufmann, 1987.
12. B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, Computer Science Department, 1996.
13. S. Marche. Measuring the Stability of Data Models. *European Journal of Information Systems*, 2(1):37–47, 1993.
14. Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems.* Object Design Inc., December 1993.
15. E.A. Rundensteiner, K.T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD'99*, pages 568–570, 1999.
16. A. Segev and J. Park. Updating Distributed Materialized Views. *IEEE Transactions on Knowledge and Data Engineering*, 1:173–184, 1989.
17. D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.
18. O_2 Technology. *O_2 Reference Manual, Version 4.5, Release November 1994.* O_2 Technology, Versailles, France, November 1994.
19. Z. Zicari. Primitives for Schema Updates in an Object-Oriented Database System: A Proposal. In *Computer Standards & Interfaces*, pages 271–283, 1991.