WPI-CS-TR-99-06 March 1999

Optimizatizing the Performance of Schema Evolution Sequences

by

Kajal T. Claypool
Chandrakant Natarajan and Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Optimizing the Performance of Schema Evolution Sequences *

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

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{kajal|chandu|rundenst}@cs.wpi.edu

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 indeed produce the equivalent final schema and data state. We also provide a 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. To validate the feasibility of our optimization approach we have implemented our optimization strategy, the CHOP optimizer, on top of the Persistent Storage Engine (PSE), the Java-based object server developed by Object Design Inc. 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, Deferred Updates, Modeling Database Dynamics, ODMG, Schema Consistency, 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. 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.

1 Introduction

1.1 Background on Schema Evolution

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 [Sjo93] 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 [Mar93], 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 [Tec94, Tec92, BMO+89, Obj93, BKKK87, Inc93].

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 [Obj93] allow users to directly edit the application source leaving the OODB system to handle the propagation of these changes to the database schema. Other systems such as Itasca [Inc93] provide a graphical user interface (GUI) that allows the users to specify their schema changes graphically while the system again propagates the schema changes to the database. Other systems such as O_2 [Tec94, Bré96], TESS [Ler96] and SERF [CJR98b] 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 [FMZ94b]. 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 [FSS+97]. 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. The focus of this work is thus to provide improvements in the execution of such sequences of schema evolution operations. While researchers have looked at deferred execution [Tec94] 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 an optimization strategy for a sequence of changes prior to execution. Our approach is orthogonal to the existing deferred execution strategy and can in fact be applied to both immediate and the deferred execution strategies [FMZ94b].

1.2 Our Proposed CHOP Approach - A Motivating Example

A schema evolution operation modifies both the class definitions and the objects associated with the extent of the modified classes. The time taken to perform the schema modification for a given schema evolution primitive is inconsequential compared to the time taken to modify the database objects. Moreover, as per Zicari et al. [FSS+97] the time for performing an object transformation is largely determined by the time taken to fetch and then later flush the page from memory [1].

Using the result from Zicari et al. [FSS+97], 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 redundant schema evolution operations.
- Optimize the execution by for example merging all schema evolution changes that operate on one extent to amortize the cost of schema evolution over several schema changes.

[1]According to Zicari et al. [FSS+97] the computation time once the page is in memory is negligible compared to the page fetch and page flush times.

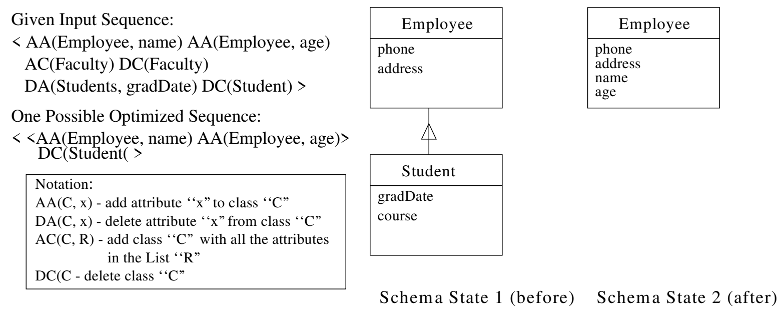

Figure 1: Sample Sequence of Schema Evolution Operations and its Optimization.

For example, the schema evolution sequence in Figure 1 will take the schema from `State 1` to `State 2`. The net structural change of the schema is the addition of the two attributes `name` and `age` to the class `Employee` and the removal of the class `Student` from the schema. In this example, it is possible to reduce this sequence of 6 operations down to an optimized sequence of 3 operations that achieve the same net structural change with better performance. For example, we can observe that the operations `AA(Employee, name)` and `AA(Employee, age)` can be **merged** as they add attributes to the same class; while the operation `DA(Student, gradDate)` can be **eliminated** by the operation `DC(Student, Employee)` as the delete of an attribute becomes redundant when the entire class is being deleted. Similarly, the operations `AC(Faculty, Employee)` and `DC(Faculty, Employee)` can be **canceled** as they are direct inverse operations thereby reducing the performance while not affecting the final state of the schema.

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. In particular, we address open issues including:

- What are the right schema evolution optimization functions for reduction?
- What are the control strategies for applying local optimization steps?
- What is the algorithm complexity for heuristic vs optimal approaches?
- What is the cost model to assess the minimality of reduced schema evolution sequences?
- Is a global minimum achievable?

1.3 Contributions

In summary, our work makes the following contributions:

- Formal analysis - a characterization of the core properties of the schema evolution operations leading to a formal basis for the CHOP optimization.
- CHOP optimizer - overall CHOP optimization strategy based including local heuristic optimization functions that reduce the sequence of schema evolution operations.
- Proof of correctness - prove that the optimized sequence of the CHOP optimizer when applied to a schema results in the *identical* schema as that produced by the application of the original sequence.
- Confluence theorem - prove that the order of application of the CHOP optimization functions is irrelevant, and thus the final optimized output sequence is guaranteed to be *unique* and *minimal.*
- CHOP implementation - prototype development of the CHOP optimizer based on the ODMG standard and PSE Pro2.0 as proof of feasibility.
- Experimental validation - confirm that the proposed optimization strategy results on average in significant performance savings and the overhead of the optimization algorithm is negligible.

Outlook. The rest of the paper is organized as follows. Section 2 presents related work while 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, i.e., the formal foundations for optimizations. Section 5 presents the actual optimization functions while Section 6 describes how these are combined to form the overall CHOP strategy. Section 8 presents our experimental evaluation. We conclude in Section 9.

2 Related Work

Current commercial OODBs such as Itasca [Inc93], GemStone [BMO+89], ObjectStore [Obj93], and O_2 [Tec94] 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 [FMZ94b, FMZ94a], 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.

In recent years, research has begun to focus on the issues of supporting more complex schema evolution operations. Breche and Lerner [Bré96, Ler96] studied the design of a set of more complex operations. Lerner [Ler96] has proposed compound type changes like *Inline*, *Encapsulate*, *Merge* etc. but more from the aspect of discovering the transformation sequences to map between these given two schemas. Lastly, our previous work on SERF [CJR98b] has provided a framework that allows the user to define arbitrarily complex schema changes by composing them out of the basic set of evolution primitives and OQL. All of these approaches for complex schema evolution transformations are based on combining the basic set of schema evolution operations. Thus, these could all potentially benefit from the CHOP optimization strategies.

Schema evolution can cause both structural as well as behavioral (code) inconsistencies. Zicari et al. [DZ91], Navathe et al. [MNJ94] and others have explored the effects of schema evolution on the methods defined for a class, while others like Bergstein et al. [BH93] have looked at it from a software perspective in terms of doing code transformations when schema evolution occurs. Ozsu et al. [OPS+95] have developed TIGUKAT, an Objectbase for modeling the uniform semantics of behaviors on objects and have explored behavioral consistency for TIGUKAT. All of these above mentioned works focus on the correctness of an individual schema evolution operation, and thus are a pre-requisite for our work. None of them addresses, however, the problem of optimizing the sequence of one or even several schema evolution operations as done by our work.

Another important issue focuses on providing support for existing applications that depend on the old schema, when other applications change the shared schema according to their own requirements. Research to address this issue has followed along two possible directions, namely, views [RLR98, RR97, Ber92] and versions [SZ86, Lau97]. To the best of our knowledge, optimization of schema evolution execution and even worse yet sequences of such operations in such multi-layered systems that would require rippling changes through possibly several different schema models remains un-explored.

3 Background - Taxonomy of Schema Evolution Primitives

The CHOP approach is based on the Object Model as presented in the ODMG Standard [Cea97]. 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. Due to space restrictions we do not discuss the ODMG object model here, rather the reader is referred to [Cea97].

Since the ODMG standard does not as yet include any standard primitives for evolution support, we have designed a taxonomy of schema evolution primitives for the ODMG object model [CJR98a] by borrowing from schema evolution primitives proposed for other data models such as O_2 or Itasca [Tec94, Tec92, BMO+89, Obj93, BKKK87, Inc93]. The taxonomy given in Table 1 is a **complete** set of schema evolution primitives such that it subsumes every possible type of schema change [BKKK87]. The taxonomy is the **essential** [Zic91] set of schema evolution primitives, i.e., each primitive updates the schema and the objects in a unique way.

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$(c,\ d)$	Rename class c to d (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$(c_x,\ a_x,\ t,\ d)$	Add attribute a_x of type t and default value d 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

Table 1: The Taxonomy of Schema Evolution Primitives

Schema evolution operations are generally categorized as **capacity-augmenting** if they increase the capacity of the schema, for instance, by adding a class, **capacity-reducing** if they decrease the capacity of the schema, for instance, by deleting a class, or **capacity-preserving** if they do not change the capacity of the schema, for instance, by changing the name of a class [RR97]. For each schema evolution primitive its capacity type is shown in the third column of Table 1.

4 Formal Foundations of Schema Evolution Sequence Analysis

To establish a foundation for our optimization principles we now develop a formal characterization of the schema evolution operations, their impact on the schema, as well as their interactions within a schema.

4.1 Properties of Schema Evolution Operations and Their Parameters

Operation Property - Properties of Schema Evolution Operations. Below we define some basic relationships for a pair of schema evolution operations `op1` and `op2` given in Table 1:

- **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. For example, `AA(Employee, age)` is **same-operation-as** `AA(Student, name)` as they both use the `add-attribute (AA)` primitive.
- **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`. For example, `AA()` is **inverse-operation-of** `DA()` as one adds an attribute and the other deletes an attribute.
- **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` [2]. For example, `DC(Employee)` also achieves `DA(Employee, age)` as part of its functionality and thus `DC(Employee)` is a **super-operation-of** `DA(Employee, age)`.

Table 2 shows how the schema evolution operations that we consider for the ODMG model (Table 1) are related by the above operation properties. Operations that are not related are said to be **independent**. As can be seen from the examples above, for optimization it is not sufficient to categorize the schema evolution

[2] We refer to `op1` as the **super-operation** and `op2` as the **sub-operation**.

	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

Table 2: 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**).

operations based on just their functionality. It is important to also know the parameters, i.e., the classes and the properties, these operations are being performed on.

Context Property - Relationships Between Schema Elements. In the analysis presented below we now extend and categorize the relationships that exist between schema elements as stated in the ODMG standard for the Schema Repository [Cea97]. Moreover, we assume that a Schema Repository for an OODB system will allow us to extract the information required to establish these relationships.

According to the ODMG standard [Cea97], each element of the Schema Repository has a **definedIn** scope which describes the scope of the element. For example, the property `Employee.name` is **definedIn** the class `Employee` with the latter describing the scope for the former.

Two classes are related by the **extendedBy** relationship if a class `A` inherits from class `B`. For example, class `Person` is **extendedBy** class `Employee` as `Employee` inherits (extends) the class `Person`.

The ODMG standard defines the notion of **sameAs** for objects based on the identity of two objects. We extend this notion of sameness to the elements of a schema, namely to classes and to properties, such that they are **sameAs** if their name and **definedIn** scope is identical. For example, $\texttt{Employee.age}_1$ is **sameAs** $\texttt{Employee.age}_2$ since $\texttt{age}_1$ is **sameAs** $\texttt{age}_2$ and they are both **definedIn** the same scope, i.e., `Employee` [3].

We also introduce an **aliasedTo** relationship between two classes or two properties such that an element is **aliasedTo** another element if the initial element is mapped to the target element through a series of name modifications. For example, class `Employee` is **aliasedTo** class `Staff` if class `Employee` is translated to class `Staff` through a series of one or more `rename-class` schema evolution primitives.

In summary we distinguish between four kinds of relationships between elements of a schema [4]:

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

4.2 Relationships between Schema Evolution Operations in a Sequence

Below we present how the properties of evolution operations and of schema elements (Section 4.1) used as their parameters can be combined to formulate the basic conditions for our optimization functions.

[3] We use the notation x_1, x_2 to distinguish two attributes that have same name for purpose of clarity.

[4] If two elements are related by the **sameAs** and the **aliasedTo** property we regard them as being the same, i.e., `property1` is **sameAs** `property2`.

Relationships between Schema Evolution Operations. Schema evolution primitives are operations that modify the schema and hence the elements of the schema, i.e., the latter provide the **context**. Combining the *operation properties* and the *context properties* defined in Section 4.1 we can develop *necessary* criteria for when an optimization function can be applied. For instance, for the two schema evolution primitives `op1` = `AA(Employee`$_1$`, name`$_1$`)` and `op2` = `DA(Employee`$_2$`, name`$_2$`)`, we know the operation property `op1` is **inverse-operation-of** `op2` holds and the context property `name`$_1$ is **sameAs** `name`$_2$ and they are **definedIn** in the same scope. As we will show later these three relations between the two operations together are *necessary* conditions that make them eligible for cancellation, i.e., they guarantee that the effect of `op1` is canceled by the execution of `op2`. However the operation and context properties alone while *necessary* are not always *sufficient* conditions for optimization.

Order-Related Properties - Order of Schema Evolution Operations in a Sequence. In some scenarios it is not sufficient to identify the parameters of two operations `op1` and `op2` to be the same but also a certain ordering among them must be satisfied. For example, if `op1` = `CCN(A, B)` and `op2` = `CCN(B, A)`, then establishing that:

- `op1` is **inverse-operation-of** `op2`,
- `A` and `B` are **definedIn** same scope and
- `A` and `B` in `op1` are **sameAs** `A` and `B` in `op2`,

while necessary is not sufficient for cancellation. The order of the parameters, i.e., the order of `A` and `B` in `op1` and `op2`, is relevant information as the reverse order of the parameters here indicates that these two operations can indeed be canceled.

When operation `op1` is **sameAs** `op2`, we identify the **schema-invariant-order** property [5]:

- 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**.

The conditions for optimization must also take into account the *relative positioning* of the operations `op1` and `op2` in the sequence. For example, consider `DA (Employee, name)` followed by `AA(Employee, name)`. Although, these two operations could be canceled [6], the database state produced by the execution of a canceled sequence (in this case a no-op) would not be the same as that produced by the original sequence. For the above example, an execution of a `DA()` followed by the `AA()` will first delete the attribute and its value from all objects. The `AA` will add a new attribute `name` and its value will be initialized to some default value for all objects. A canceled sequence however will do nothing, thus preserving the original attribute `name` and hence also all the old values for the attribute. We hence identify the following property for operations that are related by the **inverse-operation-of** property:

- **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 - Dependencies of Other Schema Evolution Operations in Sequence. The conditions for optimization must also take into account any dependencies caused by other operations that may exist between two operations in a sequence. Consider for example the sequence:

[5] All operations fall into one or the other of the two categories.

[6] They can be canceled as the operations are related by the **inverse-operation-of** property and the parameters are **sameAs** each other and are **definedIn** same scope.

< `AC(Employee, Root) AA(Employee, name) DC(Employee, Root)` >.

If `AC(Employee, Root)` and `DC(Employee, Root)` are optimized through a cancellation without considering the presence of `AA(Employee, name)`, this would produce a schema state that is not equivalent to the schema state produced by the original sequence. This leads to the **dependency** criterion.

- **Dependency Property -** The schema elements used as parameters by the two operations `op1` and `op2` being considered for optimization must not be referred by any other operation which is placed between the two operations in the sequence.

5 The CHOP Optimization Functions

5.1 An Optimization Function

Definition 1 *An optimization function F is a function that within the context of a schema evolution operation sequence Σ operates on a pair of schema evolution operations* `op1` *and* `op2` *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)), and 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.*

For the remainder of this work, we assume that the optimization function, F(`op1, op2`), is aware of the sequence Σ that `op1` and `op2` belong to as well as their respective index positions `(i)` and `(j)` in the sequence.

One 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. Towards that goal, any optimization function must observe several properties that will be 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 [BKKK87]. An important property of the optimization function therefore is for its output to also preserve the schema consistency by preserving the invariants defined for the object model.

Definition 2 (Invariant Preserving.) *An optimization function is* **invariant preserving** *if for a given invariant preserving input sequence it will produce an output sequence that is also invariant preserving, i.e., the output sequence including complex operations is equivalent to (can always be broken down into) invariant-preserving schema evolution primitives.*

Schema-State Equivalent. An important property of an optimization function is that the optimized sequence produces a schema that is the same as the schema produced by the unoptimized sequence when applied to the same input schema.

Definition 3 (Schema State Equivalent.) *Two* instances *of a schema S1 and S2 are said to be* **schema-state equivalent***, denoted by $S1 \equiv S2$, if they both have an identical schema definition, and the same number, type, content (values) and object identifiers of all the objects in the database.*

The $\equiv$ is an *equivalence* relation [AHV95] and is reflexive, symmetric and transitive, i.e., if $S1 \equiv S2$ and $S2 \equiv S3$, then $S1 \equiv S3$, $S2 \equiv S1$ and $S3 \equiv S2$. For instances of schemas the equivalence relation implies equality of the schemas. Thus if $S_0 \equiv S_1$, then $S_0 = S_1$. We now extend this equivalence to sequences of schema evolution operations.

Definition 4 (Sequence Equivalence Property.) *Two sequences of schema evolution operations, Σ_1 and Σ_2, are said to be* **sequence-equivalent***, denoted by $\Sigma_1 \equiv_\Sigma \Sigma_2$, when the two instances of schemas, S_1 and S_2, produced by the application of the two sequences on the same input schema S_0 are equivalent.*

Lemma 1 (Sequence Equivalence Relation.) *The* **sequence-equivalent** *property between two sequences of schema evolution operations* Σ_1, Σ_2, *denoted by* $\equiv_\Sigma$, *is an* equivalence *relation, i.e., it is symmetric, reflexive and transitive.*

Proof:
Here we prove that the sequence-equivalence $\equiv_\Sigma$ is reflexive, transitive and symmetric in order to prove that it is an *equivalence* relation.
Reflexive. For any sequence of schema evolution operations, Σ_1, that is applied on the input schema `S0` and produces the output schema `S1`, we know that $\Sigma_1(S0) = S1$ and $S0 \equiv S1$ and $S1 \equiv S1$ (By Definition 3). Thus, $\Sigma_1 \equiv_\Sigma \Sigma_1$, i.e., the **sequence-equivalence** relation is *reflexive.*
Symmetric. Assume two sequences of schema evolution operations, Σ_1 and Σ_2, that are sequence-equivalent as per the Definition 4. Then by Definition 4, when the two sequences Σ_1 and Σ_2 are applied to the same input schema S_0, they produce schemas S_1 and S_2 respectively, such that $S_1 \equiv S_2$. Since the schemas produced are the same and are by Definition 3 symmetric, then $S_1 = S_2$ and $S_2 = S_1$ hold. Hence, by Definition 4, $\Sigma_2 \equiv_\Sigma \Sigma_1$ also holds, i.e., $\Sigma_1 \equiv_\Sigma \Sigma_2 \Longrightarrow \Sigma_2 \equiv_\Sigma \Sigma_1$. Thus, the **sequence-equivalence** relation is *symmetric.*
Transitive. Assume 3 sequences of schema evolution operations, Σ_1, Σ_2 and Σ_3 such that $\Sigma_1(S0) = S1$, $\Sigma_2(S0) = S2$ and $\Sigma_3(S0) = S3$. Assume: $\Sigma_1(S0) \equiv_\Sigma \Sigma_2(S0)$ and $\Sigma_2(S0) \equiv_\Sigma \Sigma_3(S0)$.
We want to show that then: $\Sigma_1(S0) \equiv_\Sigma \Sigma_3(S0)$ also holds.
By Definition 4, $\Sigma_1(S1) \equiv_\Sigma \Sigma_2(S2)$, then $S1 \equiv S2$ and $\Sigma_2(S2) \equiv_\Sigma \Sigma_3(S3)$, then $S2 \equiv S3$.
By transitivity property of the $\equiv$ relation we know, $(S1 \equiv S2)$ and $(S2 \equiv S3) \Longrightarrow (S1 \equiv S3)$. Hence, by Definition 4 we know that, $\Sigma_1(S0) \equiv_\Sigma \Sigma_3(S0)$. □

Definition 5 (Schema State Equivalence Property.) *An optimization function is said to be* **schema-state-equivalence preserving** *if the input sequence,* Σ_{in}*, of the optimization function is always sequence-equivalent with the output sequence,* Σ_{out}*, produced by the optimization function .*

The **schema-state-equivalence** property for optimization functions is a key property that defines the correctness criteria for optimization functions. Thus all optimization functions abide by the following lemma.

Lemma 2 *If two optimization functions,* F_1 *and* F_2*, are schema-state-equivalence preserving then the output* Σ_1 *of one function* F_1 *is always sequence-equivalent to the output* Σ_2 *of the other function* F_2 *when both are applied to the same input sequence* Σ_{in}*.*

Proof. The proof of this can be given by the transitivity of the sequence-equivalence property shown in Lemma 1. By Definition 5, $F_1(\Sigma_{in}) \longrightarrow \Sigma_1 \Longrightarrow \Sigma_{in} \equiv_\Sigma \Sigma_1$, and $F_2(\Sigma_{in}) \longrightarrow \Sigma_2 \Longrightarrow \Sigma_{in} \equiv \Sigma_2$.
Then by transitivity of $\equiv_\Sigma$ by Lemma 1 we have $\Sigma_1 \equiv_\Sigma \Sigma_{in} \equiv_\Sigma \Sigma_2 \Longrightarrow \Sigma_1 \equiv_\Sigma \Sigma_2$. □

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`.

Definition 6 (Relative-Order Preserving.) *An optimization function is* **relative-order preserving** *if for all pairs of operations* $\texttt{op}_i$ *and* $\texttt{op}_j$ *with* $\texttt{op}_i < \texttt{op}_j$ *in the input sequence, the output sequence maintains the same relative order of the corresponding operations index(*$\texttt{op}_i$*) < index(*$\texttt{op}_j$*) assuming that* $\texttt{op}_i$ *and* $\texttt{op}_j$ *exist in the output sequence* [7]*.*

Based on the above concepts, we now extend Definition 1 to refine the definition of an optimization function.

Definition 7 (Optimization Function.) *An optimization function in CHOP is a function as defined in Definition 1 that is* **invariant-preserving** *by Definition 2,* **schema-state-equivalence preserving** *by Definition 5 and* **relative-order preserving** *by Definition 6.*

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

[7] If $\texttt{op}_i$ and $\texttt{op}_j$ are replaced by a complex operation $\texttt{op}_k$, then $\texttt{op}_k$ is at the same index position as $\texttt{op}_i$.

5.2 The Merge Optimization Function

As stated in Section 1, the time taken for performing a schema evolution operation is largely determined by the page fetch and page flush times. In our proposed CHOP approach we amortize these page fetch and flush costs over several operations by collecting all transformations on the same set of objects and performing them simultaneously [8]. 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 $<$ $\texttt{op}_1$, ... $\texttt{op}_k$ $>$, with $\texttt{k} \geq 2$. For two complex operations, op1 = $<$ $\texttt{op}_i$.... $\texttt{op}_j$ $>$ and op2 = $<$ $\texttt{op}_m$.... $\texttt{op}_n$ $>$, the operation pairs ($\texttt{op}_j$,$\texttt{op}_m$) and ($\texttt{op}_n$, $\texttt{op}_i$) are termed **complex-representative pairs**.

Definition 8 **Merge** *is an optimization function (Definition 7) 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 = $<$$\texttt{op}_i$, $\texttt{op}_j$ $>$ *and* op2 = $<$$\texttt{op}_m$, $\texttt{op}_n$ $>$*, then a relative order within the complex operations* $\texttt{op}_3$ *is maintained such that the output operation* op3 = $<$$\texttt{op}_i$, $\texttt{op}_j$, $\texttt{op}_m$, $\texttt{op}_n$ $>$. *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 the sequence in Figure 1 we can merge the operations AA(Employee, name) and the AA(Employee, age) as the two operations 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.

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 9 **Eliminate** *is an optimization function as defined in Definition 7 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:*

[8] 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.

- **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.*

For example, given the sequence in Figure 1, the operation `DC(Student`can eliminate the operation `DA(Students, gradDate)` as they meet all the requirements for elimination.

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 10 Cancel *is an optimization function as in Definition 7 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.*

Given the sequence in Figure 1 for example, the operation `AC(Faculty)` is **inverse-operation-of** `DC(Faculty)` and thus the two can be canceled.

5.5 Optimization Function Properties

5.5.1 Choosing the Right Optimization Function

In the previous sections we have presented the three optimization functions, **merge**, **eliminate** and **cancel**. However, 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 [9]. 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 the **cancel** offering the most), choosing the optimization function that offers the most reduction is very desirable. Next, we 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.

Lemma 3 (Merge Order Irrelevancy.) *An application of the optimization functions* **merge** *and* **cancel** *on one pair of schema evolution operations is equivalent to directly applying a* **cancel** *and vice versa, i.e.,* $\mathtt{C}(op_1,\ op_2) = \mathtt{C}(\mathtt{M}(op_1,\ op_2))$*. Similarly, applying a* **merge** *and* **eliminate** *on a pair of schema evolution operations is equivalent to directly applying just an* **eliminate** *and vice versa, i.e.,* $\mathtt{E}(op_1,\ op_2) = \mathtt{E}(\mathtt{M}(op_1,\ op_2))$*.*

[9]The conditions under which a **cancel** and an **eliminate** can be applied are mutually exclusive.

Proof:
We want to show that: $\texttt{MC}(op_1 \odot op_2) = \texttt{C}(op_1 \odot op_2)$ and $\texttt{ME}(op_1 \odot op_2) = \texttt{E}(op_1 \odot op_2)$ assuming cancel C and eliminate E are successful operations on $(op_1 \odot op_2)$ respectively.
MC: For the merge and the cancel functions, we can re-write the statement as follows:
$\texttt{MC}(op_1 \odot op_2) = \texttt{C}(\texttt{M}\ (op_1 \odot op_2))$
$\Longrightarrow \texttt{C}(<op_1, op_2>)$
By definition of the merge optimization function M we know that a cancel function C can be applied within the merge complex operation. Hence,
$\texttt{C}(<op_1, op_2>) = (<\texttt{C}(op_1 \odot op_2)>) = \texttt{no-op}$
Hence, $\texttt{MC}(op_1 \odot op_2) = \texttt{C}(op_1 \odot op_2) = \texttt{no-op}$.
ME: Similarly for merge and eliminate we can re-write it as follows:
$\texttt{ME}(op_1 \odot op_2) = \texttt{E}(\texttt{M}\ (op_1 \odot op_2))$
$\Longrightarrow \texttt{E}(<op_1, op_2>)$
By definition of the merge optimization function M we know that an eliminate function E can be applied within the merge complex operation. Hence,
$\texttt{E}(<op_1, op_2>) = (<\texttt{E}(op_1 \odot op_2)>) = op_1$.
Hence, $\texttt{ME}(op_1 \odot op_2) = \texttt{E}(op_1 \odot op_2) = op_1$. □

5.5.2 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 4.2 must hold, i.e., 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) are immediately optimizable 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) [10] and not be concerned about the optimization possibility between op_1 and op_3. Or we could first apply the optimization function for 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. To this end, we state the following.

Lemma 4 (Selectivity Order) *The order of selection for pairs of schema evolution operations in a sequence to which optimization functions can be applied does not prevent the achievement of* maximum *optimization.*

Proof - Proof by Induction
Base Case. For the base case we consider that the input sequence Σ_{in} is composed of three schema evolution operations op_1, op_2 and op_3, i.e.,

$$\Sigma_{in} = \mathrm{op}_1 \odot op_2 \odot op_3 \ .$$

We evaluate on a case by case basis whether the above stated lemma will hold for all possible *pair* combinations of operations ((op_1, op_2), (op_2, op_3) (op_1, op_3) [11]). We assume that all three pairs (op_1, op_2), (op_2, op_3) (op_1, op_3) are optimizable. In this scenario, op_2 can be considered as a *dependency* for the pair (op_1, op_3). For the rest of the proof we use the notation $\texttt{M}(op_1, op_2)$ to denote a pair that can be merged, $\texttt{C}(op_1, op_2)$ to denote a pair that can be canceled and $\texttt{E}(op_1, op_2)$ to denote a pair in which op_1 eliminates the operation op_2.

- Case 1. $\texttt{M}(op_1, op_2)$, $\texttt{C}(op_2, op_3)$: Given the dependency caused by op_1 we have two selection orders possible:

[10] Note that in some cases op_2 may not exist any more and hence optimizing (op_2, op_3) may no longer be possible.
[11] We use the following to denote a pair (op_1, op_2) composed of operations op_1 and op_2.

– Case 1.1. Selection Order $\mathtt{M}(op_1, op_2)$, $\mathtt{C}(op_2, op_3)$:
$\mathtt{M}(op_1, op_2)\ \mathtt{C}(op_2, op_3)[op_1, op_2, op_3\ ]$
$= \mathtt{C}(op_2, op_3)[<op_1, op_2>, op_3\ ]$
$= \mathtt{C}(op_2, op_3)[<op_1, op_2\ op_3>]$ [12]
$= ()[<op_1>]$
$= op_1$

– Case 1.2. Selection Order $\mathtt{C}(op_2, op_3)$, $\mathtt{M}(op_1, op_2)$:
$\mathtt{C}(op_2, op_3)\ \mathtt{M}(op_1, op_2)\ [op_1, op_2, op_3\ ]$
$= \mathtt{M}(op_1, op_2)[op_1\ ]$
$= op_1$

In this case the selection order of the pairs of schema evolution operations to which we can apply optimization function has been shown to be irrelevant.

- Case 2. $\mathtt{M}(op_1, op_2)$, $\mathtt{E}(op_3, op_2)\ \mathtt{C}(op_1, op_3)$: Note $\mathtt{C}(op_1, op_3)$ can only be performed when the dependency op_2 has been removed. There are two orders for selecting the optimization pairs:

  – Case 2.1. Selection Order $\mathtt{M}(op_1, op_2)$, $\mathtt{E}(op_3, op_2)\ \mathtt{C}(op_1, op_3)$:
  $\mathtt{M}(op_1, op_2), \mathtt{E}(op_3, op_2)\ \mathtt{C}(op_1, op_3)[op_1, op_2, op_3\ ]$
  $= \mathtt{E}(op_3, op_2)\ \mathtt{C}(op_1, op_3)[<op_1, op_2>, op_3\ ]$
  $= \mathtt{E}(op_3, op_2)\ \mathtt{C}(op_1, op_3)[<op_1, op_2, op_3>]$ [13]
  $= \mathtt{C}(op_1, op_3)[<op_1, op_3>]$
  $= ()[]$
  $= \mathtt{no\text{-}op}$

  – Case 2.2. Selection Order $\mathtt{E}(op_3, op_2)$, $\mathtt{M}(op_1, op_2)$, $\mathtt{C}(op_1, op_3)$:
  $\mathtt{E}(op_3, op_2)\ \mathtt{M}(op_1, op_2)\ \mathtt{C}(op_1, op_3)[op_1, op_2, op_3\ ]$
  $= \mathtt{M}(op_1, op_2)\ \mathtt{C}(op_1, op_3)[op_1, op_3\ ]$
  $= \mathtt{C}(op_1, op_3)[op_1, op_3\ ]$
  $= ()\ []$
  $= \mathtt{no\text{-}op}$

  Again all selection orders of the pairs of schema evolution operations for optimization result in the same final sequence.

- Case 3. Selection Order $\mathtt{M}(op_1, op_2)$, $\mathtt{E}(op_3, op_2)\ \mathtt{E}(op_3, op_1)$: Note $\mathtt{E}(op_3, op_1)$ can only be performed when the dependency op_2 has been removed. There are two order for selecting the optimization pairs:

  – Case 3.1. Selection order $\mathtt{M}(op_1, op_2)$, $\mathtt{E}(op_3, op_2)\ \mathtt{E}(op_3, op_1)$:
  $\mathtt{M}(op_1, op_2)\ \mathtt{E}(op_3, op_2)\ \mathtt{E}(op_3, op_1)[op_1, op_2, op_3\ ]$
  $= \mathtt{E}(op_3, op_2)\ \mathtt{E}(op_3, op_1)[<op_1, op_2>, op_3\ ]$
  $= \mathtt{E}(op_3, op_2)\ \mathtt{E}(op_3, op_1)[<op_1, op_2, op_3>]$ (By Lemma 3.)
  $= \mathtt{E}(op_3, op_1)[<op_1, op_3>]$
  $= ()[op_3\ ] = op_3$

  – Case 3.2. Selection Order $\mathtt{E}(op_3, op_2)$, $\mathtt{M}(op_1, op_2)\ \mathtt{E}(op_3, op_1)$:
  $\mathtt{E}(op_3, op_2)\ \mathtt{M}(op_1, op_2)\ \mathtt{E}(op_3, op_1)\ [op_1, op_2, op_3\ ]$
  $= \mathtt{M}(op_1, op_2)\ \mathtt{E}(op_3, op_1)\ [op_1, op_3\ ]$
  $= \mathtt{E}(op_3, op_1)\ [op_1, op_3\ ]$
  $= ()[op_3\ ]$
  $= op_3$

  In this case, also, the selection order of the pairs for optimization is irrelevant for the final optimized sequence.

[12] As per Lemma 3, we can first apply a Merge and then a Cancel and be assured that the final result is equivalent.
[13] This is by Lemma 3.

- Case 4. $\texttt{C}(op_1,\ op_2)$, $\texttt{M}(op_2,\ op_3)$: This is similar to Case 1 and hence we stipulate that the order of selection of the pairs is irrelevant in this case as well.

- Case 5. $\texttt{C}(op_1,\ op_2)$, $\texttt{E}(op_3,\ op_2)$, $\texttt{E}(op_3,\ op_1)$: The pair $\texttt{E}(op_3,\ op_1)$ can only be optimized when the dependency op_2 has been removed. Hence we consider the first two primary order of selection.

  - Case 5.1. Selection order $\texttt{C}(op_1,\ op_2)$, $\texttt{E}(op_3,\ op_2)$ $\texttt{E}(op_3,\ op_1)$:
    $\texttt{C}(op_1,\ op_2)$ $\texttt{E}(op_3,\ op_2)$ $\texttt{E}(op_3,\ op_1)$ $[op_1,\ op_2,\ op_3\ ]$
    $= \texttt{E}(op_3,\ op_2)$ $\texttt{E}(op_3,\ op_1)$ $[op_3\ ]$
    $= \texttt{E}(op_3,\ op_1)$ $[op_3\ ]$
    $= ()[op_3\ ] = op_3$
  - Case 5.2. Selection Order $\texttt{E}(op_3,\ op_2)$, $\texttt{C}(op_1,\ op_2)$ $\texttt{E}(op_3,\ op_1)$:
    $\texttt{E}(op_3,\ op_2)$, $\texttt{C}(op_1,\ op_2)$ $\texttt{E}(op_3,\ op_1)$ $[op_1,\ op_2,\ op_3\ ]$
    $= \texttt{C}(op_1,\ op_2)$ $\texttt{E}(op_3,\ op_1)$ $[op_1,\ op_3\ ]$
    $= \texttt{E}(op_3,\ op_1)$ $[op_1,\ op_3\ ]$
    $= ()\ [op_3\ ]$
    $= op_3$

In this case, also, the selection order of the pairs for optimization is irrelevant for the final optimized sequence. As can be seen from the cases above the order of selecting the pairs of operations for optimization does not prevent us from achieving the final optimized sequence.
Induction Hypothesis. Assume that for input sequence Σ_{in} composed of `k-1` operations, the selectivity order irrelevant lemma holds, i.e., for a sequence of (k-1) operations we are guaranteed to get the same final optimized sequence no matter in which order we apply the optimization functions.
Induction Step. Consider now that the input sequence Σ_{in} is composed of `k` operations, i.e., $\Sigma_{in} = op_1 \ \ldots \odot\ op_{k-1} \odot\ op_k$.
Thus we need to prove that the selectivity order irrelevant lemma holds for `k` operations, i.e., for `k` operations we will still achieve the same final optimized sequence.
There are two cases that we need to consider:

- Case 1. op_k is independent, i.e., it cannot be optimized with any of the (k-1) operations. In this case the order of selection of the pairs is irrelevant based on our induction hypothesis for the subsequence of the first `k-1` elements and hence the same final sequence if obtained. Hence the selectivity order irrelevant lemma holds.

- Case 2. op_k can form an optimizable pair with some other operation op_j, with $1 \leq \texttt{j} \leq \texttt{k-1}$, with op_j the largest `j` index that is $\leq$ `k-1`. In this case we have 2 scenarios:

  - Case 2.1. There are no dependent operations between op_j and op_k and op_j is not optimizable by any other operation, op_l, in the sequence. In this case, since op_j will not be optimized by any preceding optimization function, we know that this pair (op_j, op_k) will be optimized eventually immaterial of the order of selection. By our induction hypothesis we know that the subsequence to the left of op_j is of length $\leq$ `k-1` and the optimization of the left subsequence (op_1, op_{j-1}) and the optimization of (op_j, op_k) are independent. Hence the selectivity order irrelevant property holds for it this case as well.
  - Case 2.2. There are no dependent operations between op_j and op_k and op_j is optimizable by another operation, op_l, in the sequence with $\texttt{l} \leq \texttt{j}$. In this case, since op_j can be optimized by op_l and op_k we need to consider the order in which these pairs are picked. However, simplistically this reduces to our base case with the following three operation: $[op_l,\ op_j,\ op_k]$. We know by our base that the selectivity order irrelevant lemma holds for this.

Hence the order in which pairs of schema evolution operations are selected from a sequence of `k` operations is irrelevant in terms of obtaining the final optimized sequence. □

6 CHOP Optimization Strategy

In this section, we assume first that the CHOP optimizer 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. Below then we introduce the criteria for termination, for minimality and for optimality of this heuristic algorithm.

Definition 11 *A sequence Σ_{min} is* **minimal** *if no pair of schema evolution operations in the sequence satisfies the conditions that are set forth for either of the three optimization functions* **merge**, **eliminate** *and* **cancel**, *i.e., none of the optimization functions can be applied successfully.*

We now define the CHOP algorithm as the strategy to continuously apply optimization functions until none can be applied any more, i.e., when a minimal sequence is reached.

Definition 12 (Function Ordering.) *A series of optimization functions (***merge**, **cancel** *and* **eliminate***) applied successfully to pairs of operations in a sequence is termed a* **function ordering**.

Lemma 5 (Function Ordering Equivalence.) *The output sequence produced by a function ordering is* sequence-equivalent *to the input sequence.*

Proof - Proof by Induction
As per Definition 12, a function ordering is any combination of successful applications of the three optimization functions, merge, cancel, and eliminate. Thus a function ordering is:

$$f_1 = F_1 \odot F_2 \ldots \odot F_m,$$

where $F_i \in$ {Merge, Eliminate, Cancel} and $\forall$ i: $1 \leq \mathrm{i} \leq \mathrm{m}$.
Base Case. For the base case m = 1 we have, $f_1 = F_1$, where $F_1 \in$ {Merge, Eliminate, Cancel}. Hence $f_1(\Sigma_{in}) = F_1(\Sigma_{in}) \longrightarrow \Sigma_{f_1}$, where Σ_{f_1} is the output sequence produced by the application of the optimization function F_1, i.e., the function ordering f_1.
An optimization function by Definition 5 is schema-state-equivalence preserving, i.e., the output sequence produced by an individual optimization function is always schema-state-equivalent to the input sequence of schema evolution operations. Since the function ordering f_1 is composed of only one optimization function F_1 we have,

$$\Sigma_{in} \equiv_\Sigma \Sigma_{f_1}.$$

Induction Hypothesis. Assume that a function ordering f_{k-1} preserves the schema-state-equivalence property when it is composed of `k-1` optimization functions, i.e. if

$$f_{k-1} = F_1 \odot F_2 \ldots \odot F_{k-1}, \text{ where } F_i \in \{\text{Merge, Eliminate, Cancel}\} \text{ and i} = 1 \ldots \text{k-1}$$

and $f_{k-1}(\Sigma_{in}) = F_1 \odot F_2 \ldots \odot F_{k-1}(\Sigma_{in}) = \Sigma_{f_{k-1}}$, where $\Sigma_{f_{k-1}}$ is the output sequence produced by the application of `(k-1)` optimization functions i.e., the function ordering f_{k-1}, then we have $\Sigma_{f_{k-1}} \equiv \Sigma_{in}$.
Induction. Consider that the function ordering is now composed of `k` optimization functions, i.e.,

$$f_k = F_1 \odot F_2 \ldots \odot F_{k-1} \odot F_k, \text{ where } F_i \in \{\text{Merge, Eliminate, Cancel}\} \text{ and i} = 1 \ldots \text{k}.$$

We know $f_k = f_{k-1} \odot F_k$.

Re-writing the above statement such that F_k is an optimization function that is applied on the result of the function ordering f_{k-1} we have,

$$f_k(\Sigma_{in}) = F_k(f_{k-1}(\Sigma_{in}))$$
$$\Longrightarrow f_k(\Sigma_{in}) = F_k(\Sigma_{f_{k-1}})$$
$$\Longrightarrow f_k(\Sigma_{in}) = \Sigma_{f_k}$$

Since Σ_{f_k} is produced by the application of one optimization function, F_k, on the sequence, $\Sigma_{f_{k-1}}$, by Definition 5, we have $\Sigma_{f_k} \equiv_\Sigma \Sigma_{f_{k-1}}$. We know from our induction hypothesis that $\Sigma_{f_{k-1}} \equiv_\Sigma \Sigma_{in}$. Thus by transitivity (Lemma 1) we have $\Sigma_{f_k} \equiv_\Sigma \Sigma_{in}$.

This implies that the sequence produced by any function ordering is sequence-equivalent with the input sequence. □

Definition 13 (Complete Function Ordering.) *A function ordering is defined to be a* **complete function ordering** *when its application to an input sequence Σ_{in} yields a* **minimal** *sequence Σ_{min} (Definition 11).*

As each of the optimization functions either reduces the number of operations in a sequence or compounds them into complex operations and none of them undoes the effects of the other, the maximal length of a complete function ordering is O(n), where `n` is the length of the input sequence. However, given that there can potentially be a vast number of complete function orderings, the search space to enumerate all of the orderings and select an **optimal** function ordering is exponential. In fact, for a sequence of length `n` the number of possible complete function orderings is of $O(n^n)$.

In order to select an optimal function ordering we consider an enumeration of all the possible complete function orderings for a given input sequence as a tree with each application of an optimization function represented by a child node. The depth of each branch of the tree could then be indicative of the cost of the schema evolution itself. Thus, one heuristic for choosing an optimal complete function ordering could potentially be based on the depth of the branch. Another heuristic could base the selection on the number of operations in the final output sequence. The problem with these heuristic selection criteria is however the cost associated with the search space. One important result of our work addresses this limitation and finds that all complete function orderings conform to the confluence property as defined in term rewriting systems [Gra98], i.e., all complete function orderings converge to one **unique minimal** sequence.

Theorem 1 [Confluence Theorem]: *Given an input schema evolution sequence, Σ_{in}, all complete function orderings f_i produce minimal resultant sequences Σ_i that are all exactly the same.*

Proof - Proof by Contradiction
Given:
$f_0(\Sigma_{in}) \longrightarrow \Sigma_0$
$f_1(\Sigma_{in}) \longrightarrow \Sigma_1$
$\vdots$
$f_n(\Sigma_{in}) \longrightarrow \Sigma_n$

where f_0, f_1 ... f_n are complete function orderings applied on the input sequence Σ_{in} that produce the minimal sequences Σ_0, Σ_1, ...,Σ_n respectively, such that $\Sigma_{in} \equiv_\Sigma \Sigma_0$, $\Sigma_{in} \equiv_\Sigma \Sigma_1$, etc. By the transitivity of the sequence-equivalence relation (Lemma 1), we have $\Sigma_{in} \equiv_\Sigma \Sigma_0 \equiv_\Sigma \Sigma_1 \equiv_\Sigma \ldots \equiv_\Sigma \Sigma_n$. This in turn implies by Definition 4 that the schemas produced by these sequences are also equivalent, i.e., $S_0 \equiv S_1 \ldots \equiv S_n$.

The input and the output sequences are by definition composed of either primitive schema evolution operations or complex operations. Thus, for example, $\Sigma_{in} = \text{op}_1 \odot \text{op}_2 \ldots \odot op_n$, where $op_1 \ldots op_n$ are schema evolution operations or complex operations . We use $|\Sigma_{in}|$ to denote the length of the sequence, i.e., the number of operations in a sequence. Hence, $|\Sigma_{in}| = \mathtt{n}$.
Worst Case. Assume no optimization is possible, i.e., Σ_{in} is a minimal sequence. This implies:

$$\Sigma_{in} = \Sigma_0 = \Sigma_1 = \ldots = \Sigma_n.$$

Thus, in this case, there is no optimization possible and all complete function orderings, f_i, produce the same minimal resultant sequence as the input sequence, i.e., $\Sigma_i = \Sigma_{in}$.
Best Case. In this case, there is complete optimization, i.e., the input sequence Σ_{in} is reduced to a sequence, Σ_i, such that $|\Sigma_i| = 0$, by some complete function ordering f_i. By the transitivity as per Lemma 1,

$$\Sigma_{in} \equiv_\Sigma \Sigma_0 \equiv_\Sigma \Sigma_1 \ldots \Sigma_{i-1} \equiv_\Sigma \Sigma_i \ldots \equiv_\Sigma \Sigma_n$$

Considering $\Sigma_i \equiv_\Sigma$`no-op` and $\Sigma_i \equiv_\Sigma \Sigma_j$, for some $1 \leq i,j \leq n$ and $i \neq j$. However, as Σ_j and Σ_i are equivalent and they produce the same schema, i.e., $S_j = S_i$, the net effects of the application of Σ_j must also be null (`no-op`). But given that Σ_j has been optimized we can assume that Σ_j does not contain any more optimizable operations and hence, $\Sigma_i = \Sigma_j$. Extending this result to the sequences produced by other function orderings, we have:

$$\Sigma_0 = \Sigma_1 = \ldots = \Sigma_n = \texttt{no-op}.$$

Average Case. In the average case, we have some optimization. Consider two sequences Σ_i and Σ_j produced by two complete function orderings f_i and f_j respectively and $|\Sigma_{in}| = n$, then we have, $0 < i,\ j < n$, where $|\Sigma_i| = i$ and $|\Sigma_j| = j$ respectively. There are three possibilities:

- $i = j$. We know that the two sequences, Σ_i and Σ_j, produce the same schema, i.e., $S_i = S_j$ (by Definition 4) and the two output sequences are composed of *essential* schema evolution primitives. By Lemmas 3 and 4, we are assured that if there was a schema evolution operation pair that could be optimized then it would be optimized independent from the order in which the pairs themselves were selected. By Lemma 4 we are assured that the final outcome is the same, irrelevant of the intermediate sequences. Hence, if two sequences are of the same length and are equivalent, then by Lemma 4 we can say that the two sequences cannot be a permutation of each other and thus are the same.

- $i > j$. Here as before, $\Sigma_i \equiv_\Sigma \Sigma_j$, and thus by Lemma 1 we have $S_i = S_j$. However, in this scenario we have the number of operations in the sequence $|\Sigma_i| > |\Sigma_j|$. As the optimization functions are composed of the **essential** set of schema evolution primitives, we assume that all of the schema evolution operations that exist in the sequence Σ_j must also exist in Σ_i, thus $\Sigma_j \subset \Sigma_i$.

  - The extra operations in Σ_i ($\Sigma_i - \Sigma_j$) are *independent* operations, i.e., they act on a different part of the schema and hence can not be optimized by any of the optimization functions (by the minimality of the function ordering). In this case, however, this would produce a different schema than that produced by Σ_j, which violates the sequence equivalence (Definition 4).
  - The extra operations in Σ_i ($\Sigma_i - \Sigma_j$) are *redundant* operations, i.e., their combined effect is a `no-op`. However, this implies that at the very least the **cancel** or an **eliminate** optimization function can be applied to the sequence. This implies that the sequence can be optimized (reduced) further. However, this contradicts our basic premise that the sequence is **minimal**.

- $j > i$. The same argument as above holds for this as well with the order of `i` and `j` now reversed..

In summary, in addition to all the sequences produced by complete function orderings being sequence equivalent, they are also **equal**, i.e., $\Sigma_0 = \Sigma_1 \ldots \Sigma_i = \Sigma_j \ldots = \Sigma_n$. □

This theorem reduces the complexity of our problem of finding an optimal output sequence from exponential to polynomial. Based on just simple heuristics as discussed before, it is hard to guarantee that the chosen output sequence is *better* in terms of execution performance than another output sequence. As per the theorem, however, it is no longer necessary to enumerate all possible function orderings to find the **optimal minimal** sequence. The CHOP optimizer can simply choose any one of all possible function orderings and it is always guaranteed that the final output sequence is globally **minimal**.

7 The CHOP Implementation

CHOP Requirements. The implementation of CHOP assumes that certain features are provided by the underlying OODB system. Below we list these required features:

- **Persistence capability:** CHOP requires the OODB to be capable of storing data objects persistently and retrieving them when necessary. This feature is essential and is provided by all existing OODB systems.

- **Extensibility**: Since we are building CHOP as a layer above the OODB system, we require the OODB architecture to be modular enough to facilitate this.

- **Modular Schema Evolution Manager layer:** A very crucial requirement for CHOP to be effective is that the schema evolution manager must provide a separation between schema-level updates and object-level updates. This is essential for the merge optimization as that hinges on the capability of performing all the schema-level updates at one time followed by the object-level updates. Hence, CHOP needs to be able to hold back the object-level updates until all the schema-level updates for a merged complex operation have been performed. This capability to control the time when the object-level update for each SE operation is performed is thus a very important requirement that the OODB must satisfy. Most OODBs do not provide this capability as a built-in feature as it is not needed by end users, but since several of them are extensible, we expect that this feature could be easily provided by exposing appropriate APIs.

CHOP Architecture. Figure 2 gives the general architecture of the CHOP system composed of three layers. The top layer contains the *User Interface* which supplies the user input in the form of a sequence of schema evolution operations. The middle layer corresponds to the CHOP optimizer as presented in this paper, while the bottom layer represents system components that we expect any underlying OODB system to provide. For our implementation platform we have used PSE Pro 2.0 running on Windows NT as the underlying OODB system [O'B97]. PSE Pro 2.0 is the first persistent storage engine written entirely in Java [O'B97] and runs within the same process as the Java applications or applets. The PSE Java client and the storage layer provide an easy-to-use interface for storing and retrieving persistent objects.

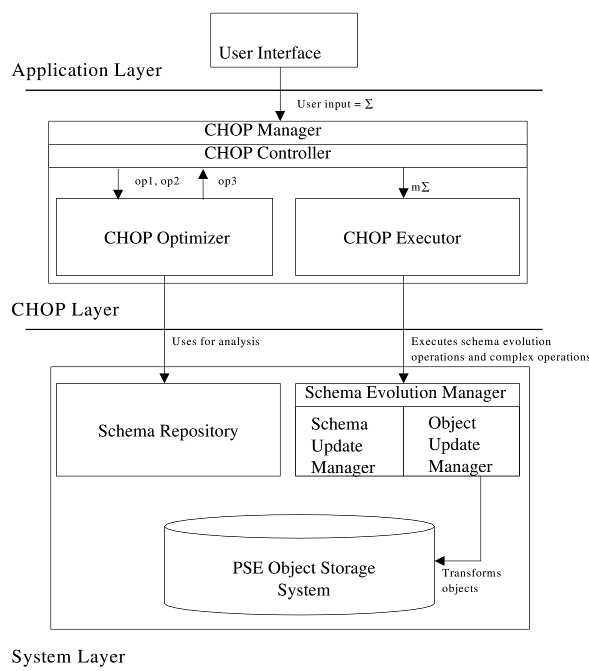

Figure 2: System Architecture of CHOP Implementation

CHOP Tool. The user input to the CHOP system is a sequence of schema evolution operations. The CHOP Manager manages the sequence of schema evolution operations, provides access to a pair of schema evolution operations to be used as input by the optimization functions, and also manages the correct placement of the output schema evolution operation in the sequence [14].

The CHOP Manager invokes the CHOP Controller which incorporates a control strategy for controlling the execution of the optimization function on the sequence. The CHOP Controller is also responsible for the

[14]As stated earlier, by default we place the output schema evolution operation at the index position of the first input operation. The index of the second input operation is set to an empty operation by the CHOP Manager.

termination of the optimization. In general termination is determined when no optimization function can be applied successfully for any pair of schema evolution operations in the sequence. The CHOP Controller invokes one of the optimization functions and passes the pair of operations as parameters to the invoked function.

The CHOP Optimizer is responsible for correctly applying the optimization function selected by the CHOP Controller on a given pair of schema evolution operations.

To encapsulate the notion of complex schema evolution operations within CHOP, we have built the CHOP Executor that controls the execution of the complex operations by invoking the correct primitives through the Schema Evolution Manager.

OODB System Components. The Schema Evolution Manager provides an interface for the execution of the set of schema evolution primitives as described in Section 3. It interacts with the Schema Repository that we have built on top of the PSE Pro 2.0. More details of the actual implementation can be found in [CJR98a]. The Schema Repository contains information on each class and its placement in the class hierarchy. It is also responsible for the migration of objects from the existing (old) class definition to the changed (new) class definition, thus keeping them updated and consistent with the schema change. For CHOP, we have enhanced this schema evolution manager such that it separates it into two processes and provides separate APIs for both schema updates and object updates. We distinguish these as Schema Update Manager and Object Update Manager. This allows us to perform several schema-level updates on a class before applying these changes to the affected objects.

8 Experimental Validation

We have conducted several experiments to not only evaluate the potential performance gains of the CHOP optimizer but also to experimentally validate the confluence property of complete function orderings.

Experimental Setup. Our experimental system, CHOP, was implemented as a pre-processing layer over the Persistent Storage Engine (PSE Pro2.0). Our work is based on the ODMG object model and for this we have built an ODMG-compliant Schema Repository and a dynamic schema evolution facility for the ODMG object model using PSE Pro2.0 as our database (For more implementation details see [CJR98a]).

The input to the CHOP system are sequences of schema evolution operations from the set defined in Section 3. Given our result of *confluence*, the control strategy for picking a pair of operations from the input sequence is based on a linear traversal of the sequence. We use the `payroll` schema as shown in Figure 3 as the basis for building the input sequences of schema evolution operations. This schema is populated with 5000 objects per class. Due to lack of availability of a benchmark of typical sequences of schema evolution operations, the input sequences themselves were randomly generated sequences. All experiments were conducted on a Pentium II, 400MHz, 128Mb RAM running WindowsNT and Linux.

Baseline Experiments - One Operation. To establish a baseline for our experiments we performed two tests to measure the unoptimized schema evolution processing time on our test database. The first test measures the time needed for the schema evolution process while varying the number of attributes in a class. The second test measures the time required by the schema evolution process as the number of database objects in the class are varied.

For the first test, the number of objects and the schema evolution operations itself were kept at a constant. Figure 4 shows the experiment results for add_attribute operation and an extent size of 100 objects. Figure 5 shows the same experiment with the number of objects now constant at 1000. We notice that as the number of attributes increases, the curve showing the time requirement increases non-linearly with slightly increasing slope. These results confirm that the number of objects has a very minimal effect on the general trend for attribute v/s time variation. The slight deviation from a linear nature of the slope is due to system factors such as page faults, etc., which occur as the size of the objects grows.

For the second test we kept the number of attributes in the referred class constant at 4 (see Figure 6). Here , we measure the cost of performing one schema evolution operation on a varying number of objects. We observe that the curve is linear. This is in accordance with our expectations as well as observations

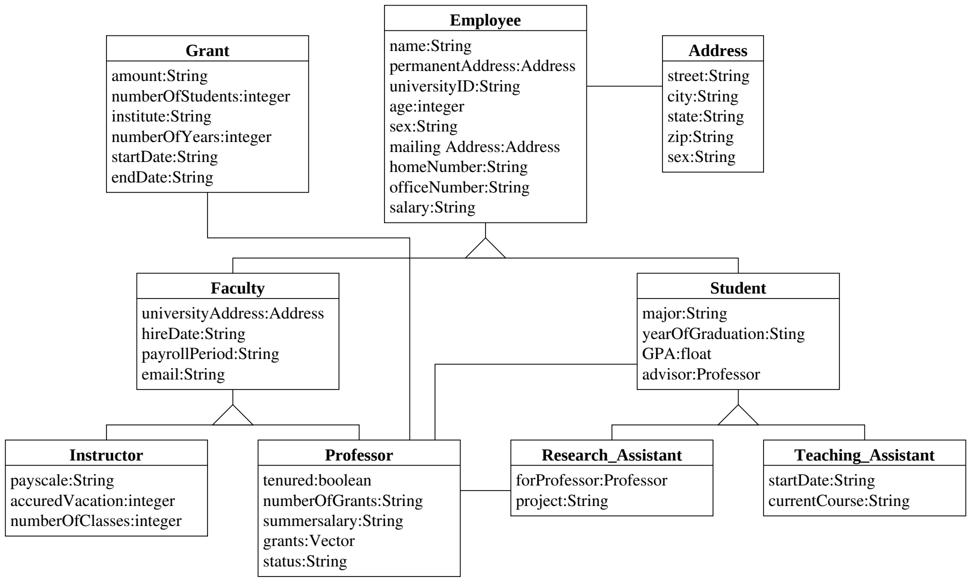

Figure 3: Example Schema Used for Experiments - Payroll Schema (UML Notation)

by others in the literature [Mey92]. Hence the schema evolution processing time for a given class increases linearly with an increase in the number of objects affected by the schema change.

Thus, the schema evolution processing time for a sequence is proportional to the number of objects in the schema. The time needed per SE operation increases linearly with an increase in the number of attributes in the referred class. Hence, for larger databases we can potentially have larger savings.

Optimized vs Unoptimized Sequences. Key experiment to show how our strategy can indeed provide performance was a direct comparison between the performance of an optimized vs an unoptimized sequence. Here, we have limited our sample schema to four classes with four attributes each. Figure 7 shows the optimized and unoptimized SE processing times for a sequence of fixed length. While the input sequence had eight SE operations, the optimized sequence had three SE operations. Figure 8 shows the same two curves for another SE operation sequence with five operations. Here the optimized sequence reduces to a single complex operation containing 2 schema evolution operations. Both these graphs incorporate the cost

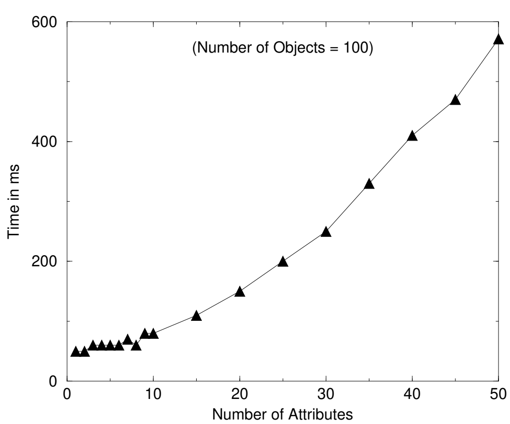

Figure 4: SE Processing Time v/s Number of Attributes (1)

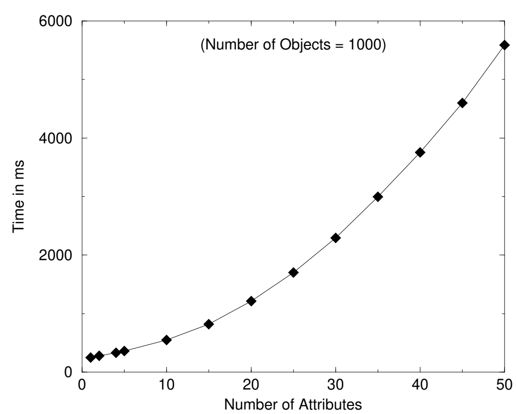

Figure 5: SE Processing Time v/s Number of Attributes (2)

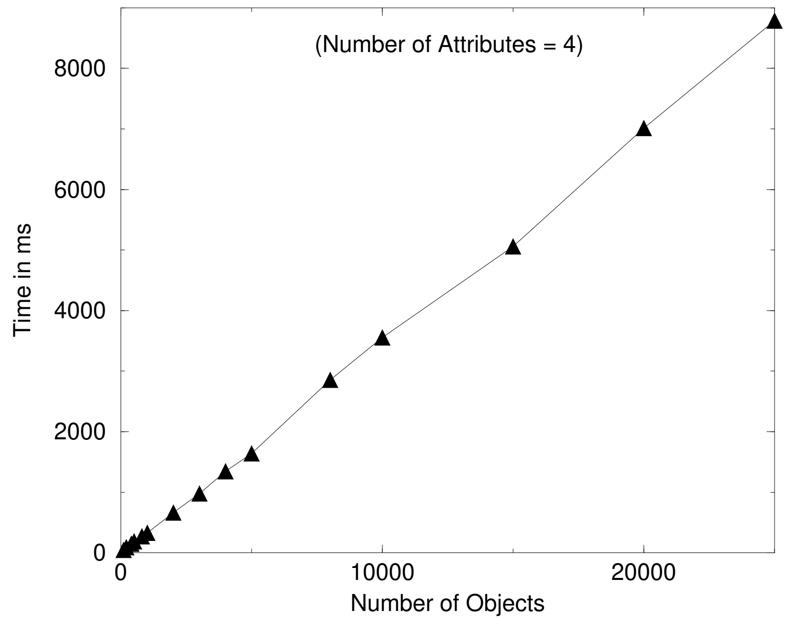

Figure 6: SE Processing Time v/s Number of Objects

of *CHOP* execution, i.e., the algorithm overhead.

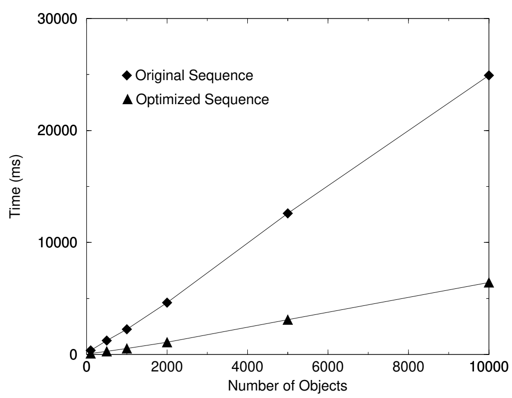

Figure 7: Sequence Execution Time for Optimized v/s Un-optimized Sequence with Algorithm Overhead on the Sample Schema

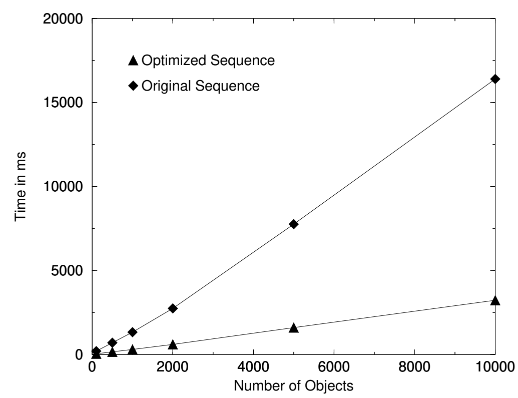

Figure 8: Sequence Execution Time for Optimized v/s Un-optimized Sequence with Algorithm Overhead on the Sample Schema

Performance Gains. In order to get an estimate of the performance gains of CHOP we have measured the execution time for optimized versus unoptimized input sequences. For a set of random input sequences we run CHOP to produce their respective minimal sequence. In our experiments, we encountered sequences that reduced to length zero, length one, etc. all the way to no reduction at all. Figure 9 shows the schema evolution processing times for different output sequences for an initial input of length 8. The execution time for a sequence decreases in uniform ratios as the number of operations in the output sequence decreases. An optimized length of zero represents the best case while a sequence with the optimized length of eight represents the worst case.

The chart in Figure 10 shows a more detailed view of the best and worst case conditions taken from

Figure 9 with the CHOP algorithm overhead time taken into consideration. As seen from these curves, the CHOP pre-processing optimization overhead is negligible in both best and worst case conditions (59ms and 78ms respectively). Thus there is no loss in terms of overhead even for cases where CHOP is not able to reduce the input sequence at all while the potential performance savings are immense.

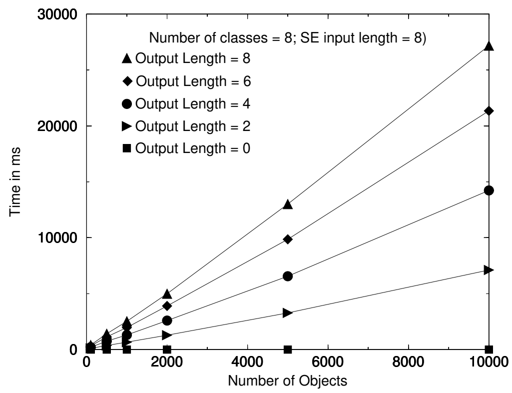

Figure 9: Sequence Times w/o Algorithm Overhead for Input Sequences of Length 8 on the Sample Schema.

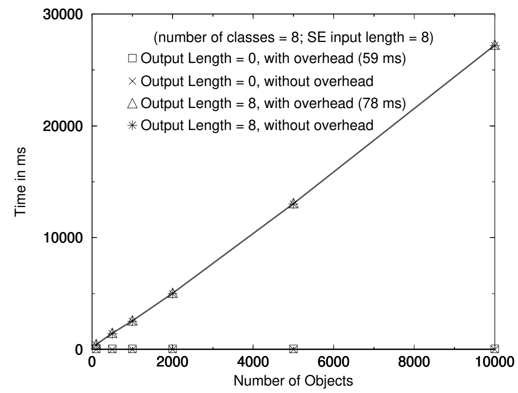

Figure 10: Best and Worst Case Sequence Times with Algorithm Overhead for Input Sequences of Length 8 on the Sample Schema.

Average Degree of Optimization. In short, the performance gain achievable by CHOP is proportional to the degree to which the input sequence of schema evolution operations can be reduced in length. Thus, the **degree of optimization** of a sequence is the ratio given by the initial sequence versus the reduced sequence length. Figures 11 and 12 show two bar charts depicting the optimizations obtained for randomly generated sequences of length 8 with the number of classes in the schema varying from 8 to 16. The graphs depict the distribution of optimization (with respect to the number of schema evolution operation in the output sequence) for 50 sequences of length 8. With the sequence lengths equal to the number of classes on which schema evolution operations were being applied, we were on average able to get better optimization. However, as the schema evolution operations were spread over a large number of classes, i.e., the number of classes was larger than the length of the sequence of schema evolution operations, the optimization in terms of the reduction was on average not as good. It can be observed that on average the degree of optimization increases with the divergence of the number of classes in a schema and the number of schema evolution operations in a sequence.

Summary.

Summary of Results. We have presented here a representative set of our experiments. For more details on these experiments please refer to [Nat98]. To summarize, we have found the following results:

- The SE processing time for a sequence is proportional to the number of objects in the schema. The time needed per SE operation increases super-linearly with an increase in the number of attributes in the referred class. Hence, for larger databases we can potentially have larger savings.
- The schema evolution processing time is linearly proportional to the length of the sequence, hence the more evolution operations are reduced, the larger the 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.
- 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.

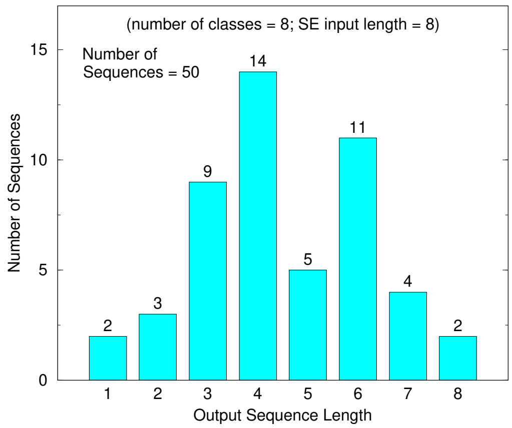

Figure 11: Distribution: Number of Classes = Sequence Length

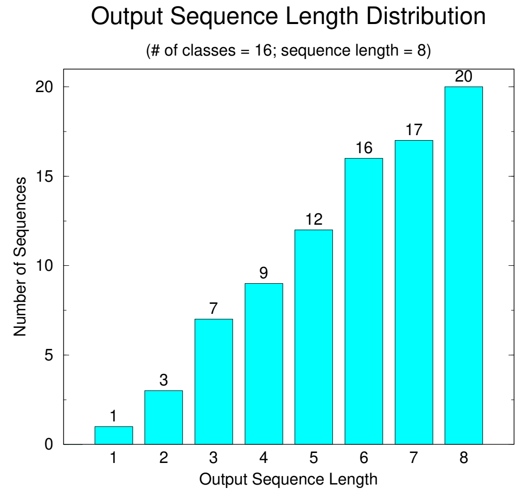

Figure 12: Distribution: Number of Classes > Sequence Length

- 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.

9 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 will be presented as a demo at SIGMOD'99 [RCL+99]. 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.

While the CHOP optimizer was initially developed for OODB schema evolution optimization, much of the analysis as well as the optimizations suggested in this paper can be applied to other domains such as Data Warehousing to optimize the schema changes reported to the warehouse by the information source or for complex database transformations that perhaps contain queries inter-leaved with the schema evolution operations [CJR98b].

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research. In particular, we are grateful to Prof. George T. Heineman for his reviews and feedback. We would also like to thank Jing Jin, Anuja Gokhale, Parag Mahalley, Swathi Subramanian and Jayesh Govindrajan for their input on the implementation of schema evolution support for PSE Pro2.0.

References

[AHV95] S. Abiteboul, R. Hull, and Vianu V. *Foundations of Databases.* Addison-Wesley Publishing Company, 1995.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In *3rd Int. Conference on Extending Database Technology*, pages 136–151, March 1992.

[BH93] P.L. Bergstein and W.L Hursch. Maintaining Behavioral Consistency during Schema Evolution. In *International Symposium on Object Technologies for Advanced Software*, pages 176–193, 1993.

[BKKK87] 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.

[BMO+89] 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.

[Bré96] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

[Cea97] R.G.G. Cattell and et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Inc., 1997.

[CJR98a] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.

[CJR98b] 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.

[DZ91] C. Delcourt and R. Zicari. The Design of an Integrity Consistency Checker (ICC) for an Object Oriented-Database System. In P. America, editor, *ECOOP*, pages 97–117, 1991.

[FMZ94a] 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.

[FMZ94b] 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.

[FSS+97] C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, S. Zaniolo, C. Ceri, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.

[Gra98] B. Gramlich. On Termination and Confluence Properties of Disjoint and Constructor-Sharing Conditional Rewrite Systems. *Theoritical Computer Science*, 165(1):97–131, 1998.

[Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

[Lau97] S.-E. Lautemann. Schema Versions in Object-Oriented Database Systems. In *Int. Conference on Database Systems for Advanced Applications (DASFAA)*, pages 323–332, 1997.

[Ler96] 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.

[Mar93] S. Marche. Measuring the Stability of Data Models. *European Journal of Information Systems*, 2(1):37–47, 1993.

[Mey92] B. Meyer. Applying "Design By Contract". *IEEE Computer*, 25(10):20–32, 1992.

[MNJ94] M.A Morsi, S. Navathe, and Shilling J. On Behavioral Schema Evolution in Object-Oriented-Database System. In *Int. Conference on Extending Database Technology (EDBT)*, pages 173–186, 1994.

[Nat98] C. Natarajan. CHOP: An Optimizer for Schema Evolution Operation Sequences. Master's thesis, Worcester Polytechnic Institute, June 1998.

[O'B97] P. O'Brien. Making Java Objects Persistent. *Java Report*, 1(1):49–60, 1997.

[Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems.* Object Design Inc., December 1993.

[OPS+95] M.T. Oszu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A Uniform Behavioral Objectbase Management System, 1995.

[RCL+99] 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*, 1999.

[RLR98] E.A. Rundensteiner, A. Lee, and Y.-G. Ra. Capacity-Augmenting Schema Changes on Object-Oriented Databases: Towards Increased Interoperability. In *Object-Oriented Information Systems*, 1998.

[RR97] Y. G. Ra and E. A. Rundensteiner. A Transparent Schema Evolution System Based on Object-Oriented View Technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, September 1997.

[Sjo93] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.

[SZ86] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Databases. In *Proc. 1st OOPSLA*, pages 483–494, 1986.

[Tec92] Versant Object Technology. *Versant User Manual.* Versant Object Technology, 1992.

[Tec94] O_2 Technology. *O_2 Reference Manual, Version 4.5, Release November 1994.* O_2 Technology, Versailles, France, November 1994.

[Zic91] Z. Zicari. Primitives for Schema Updates in an Object-Oriented Database System: A Proposal. In *Computer Standards & Interfaces*, pages 271–283, 1991.