# A Transparent Approach for Database Schema Evolution Using View Mechanism

Jianxin Xue, Derong Shen, Tiezheng Nie, Yue Kou, and Ge Yu

College of Information Science and Engineering, Northeastern Unicersity, China
xuejianxin@research.neu.edu.cn,
{shenderong,nietiezheng,kouyue,yuge}@ise.neu.edu.cn

**Abstract.** Designing databases that evolve over time is still a major problem today. The database schema is assumed to be stable enough to remain valid even as the modeled environment changes. However, the database administrators are faced with the necessity of changing something in the overall configuration of the database schema. Even though some approaches proposed are provided in current database systems, schema evolution remains an error-prone and time-consuming undertaking. We propose an on-demand transparent solution to overcome the schema evolution, which usually impacts existing applications/queries that have been written against the schema. In order to improve the performance of our approach, we optimize our approach with mapping composition. To this end, we show that our approach has a better potential than traditional schema evolution technique. Our approach, as suggested by experimental evaluation, is much more efficient than the other schema evolution techniques.

**Keywords:** schema evolution, schema mapping, backward compatibility, schema composition.

## 1 Introduction

Database designers construct schemas with the goal of accurately reflecting the environment modeled by the database system. The resulting schema is assumed to be stable enough to remain valid even as the modeled environment changes. However, in practice, data models are not nearly as stable as commonly assumed by the database designers. As a result, modifying the database schema is a common, but often troublesome, occurrence in database administration. These are significant industrial concerns, both from the viewpoint of database system manufacturers and information system users. Schema evolution, and its stronger companion, schema versioning, have arisen in response to the need to retain data entered under schema definitions that have been amended. A more formal definition of schema evolution is the ability for a database schema to evolve without the loss of existing information [1].

**Motivating Example.** To better motivate the need for schema evolution support, we illustrate a schema evolution example in an employee database, which is used as a running example in the rest of the paper. Fig. 1 outlines our example, which has three schema versions, $V_1$ through $V_3$.

Due to a new government regulation, the company is now required to store more personal information about employees. At the same time, it was required to separate employees' personal profiles from their business-related information to ensure the privacy. For these reasons, the database layout was changed to the one in version $V_2$, where the information about the employees is enriched and divided into two tables: Personal, storing the personal information about the employees, and Employees, maintaining business-related information about the employees.

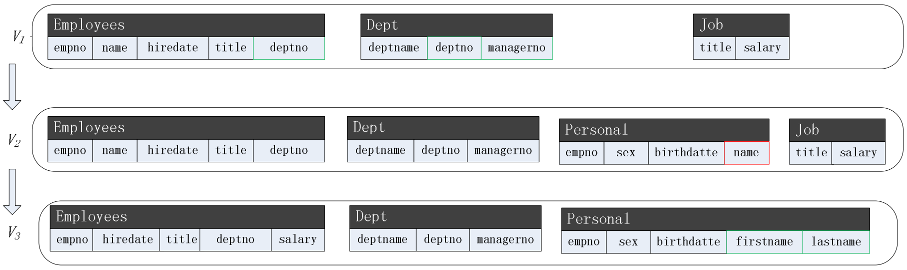

**Fig. 1.** Schema evolution in an employee database

The company chose to change its compensation policy: to achieve 'fair' compensation and to better motivate employees, the salaries are made dependent on their individual performance, rather than on their job titles. To support this, the salary attribute was moved to the table employees, and the table job was dropped. Another modification was also introduced that the first name and last name are now stored in two different columns to simplify the surname-based sorting of employees. These changes were represented as the last schema version, $V_3$.

Assume that a department works for a project, and view E_P is used to correlate the employees with the projects they work for. On top of this view, we consider that the report module contains an aggregate query that calculates the expenses of the project per month by summing up the salaries of all employees working for it and compare them with the budget of the project. Along with the evolution of the schema, the view E_P becomes unavailable. How to effectively keep the application available is our object.

**Contribution.** Our main contributions in this paper are summarized as follows.

1. A novel on-demand schema evolution approach based on virtual views (named as on-demand approach) is proposed, which only deals with the schemas effected by the schema evolution and preserves the new data instead of the version data.
2. Subsequently we construct view version with mapping inversion, which can recover all the source information with complementary approach.
3. Next, an optimization algorithm is presented to improve application/queries efficiency.

4. At last, we have performed comprehensive experiments to compare our approach with traditional database schema evolution techniques, which show that our approach achieves remarkable improvement in efficiency.

The remainder of this paper is organized as follows: Section 2 discusses related works, Section 3 introduces preliminary definitions, Section 4 discusses in details the design of our schema evolution method, Section 5 is dedicated to experimental results. We conclude in Section 6 where we also state our future work.

## 2 Related Works

Schema evolution means modifying schemas within a database without loss of existing data. With the acceleration of database schema modification frequency of Internet and enterprise, database schema evolution becomes a hotspot in current database research. However, current database schema evolution is mainly implemented manually.

Schema evolution has been extensively addressed in the past and a variety of techniques have been proposed to execute change in the least erratic way possible to avoid disruption of the operation of the database. A bibliography on schema evolution lists about 216 papers in the area [2]. The proposed solutions for schema evolution can be categorized mainly by following one of these approaches: modification, versioning and views.

**Modification.** The original schema and its corresponding data are replaced by a new schema and new data. This approach does not exactly adhere to the schema evolution definition, which makes the applications that use the original schemas inconsistent with the new database schemas [3]. This renders the approach unsuitable in most real cases, yet it remains the most popular with existing DBMS. In [4], the authors discuss extensions to the conventional relation algebra to support both aspects of evolution of a database's contents and evolution of a database's schema. In [5], authors present an approach to schema evolution through changes to the ER schema of a database. In [6], they describe an algorithm that is implemented in the $O_2$ object database system for automatically bringing the database to a consistent state after a schema update has been performed.

**Versioning.** The old schema and it corresponding data are preserved and continued to be used by existing applications, but a new version of the schemas is created, which incorporates the desired changes [7]. There are two most used versioning methods. The first is sequential revisions, which consists of making each new version a modification of the most recent schema. This approach is adopted in Orion Database System [8]. The second method is a complex one, which is called parallel revisions. For instance, Encor Database System[9]. Generally, the versioning approach presents performance problems.

**View.** A view is a derived table. It makes possible to change the schema without stopping the database and destroying its coherence with existing applications.

Bellahsene [10] proposes a method that uses view to simulate the schema changes and the data migration is not needed, i.e., views are viewed as the target schema. However, this method has scalability limitations. In fact, after several evolution steps, the applications/queries may involve long chains of views and thus deliver poor performance. Moreover, it is difficult to change current schema.

In [11], the authors deal with the adaption of the view definition in the presence of changes in the underlying database schema. [12], [13] deal also with a specialized aspect of the view adaptation problem. The work of [14] employs a directed graph for representing the object dependencies in O-O database environments and finding the impact of changes in database objects towards application objects.

As of today, in [15], the authors tend to extend the work of [16]. They first consider a set of evolution changes occurring at the schema of a data warehouse and provide an informal algorithm for adapting affected queries and views to such changes. The most representative achievement is PRISM developed by Curino. One contribution of the work on PRISM is a language of Schema Modification Operator. Meanwhile, automatic query rewriting of queries specified against schema version *N* into semantically equivalent queries against schema Version *N+1*, and vice versa [17]. SMOS have a good semantic express for schema evolution, but still has some limitations.

## 3 Preliminaries

A schema R is a finite sequence $(R_1,\ldots,R_k)$ of relation symbols, where each $R_i$ has a fixed arity. An instance *I* over *R* is a sequence $(R^i{}_1,\ldots,R^i{}_k)$ , where each $R^i{}_i$ is a finite relation of the same arity as $R_i$. We shall often use $R_i$ to denote both the relation symbol and the relation $R^I{}_i$ that instantiates it. We assume that we have a countable infinite set Const of constants and a countably infinite set Var of labeled nulls that is disjoint from Const. A fact of an instance *I* is an expression $R^i{}_i(v_1,\ldots,v_m)$ where $R_i$ is a relation symbol of *R* and $v_1,\ldots,v_m$ are constants or labeled nulls such that $(v_1,\ldots,v_{\mathrm{m}})\in R^i{}_i$. The expression $v_1,\ldots,v_m$ is also sometimes referred to as a tuple of $R_i$. An instance is often identified with its set of facts.

A schema mapping is a triple $M=(S,T,\Sigma)$, where *S* is the source schema, *T* is a target schema, and $\Sigma$ is a set of constraints that describe the relationship between *S* and *T*. *M* is semantically identified with the binary relation:

$$Inst(M)=\{(I,J)|\, I\in S_i\,,\, J\in T_i,(I,J)|= \Sigma\}. \tag{1}$$

Here, $S_i$ is the instance of source schema and $T_i$ is the instance of target schema. We will use the notation (I, J)|= $\Sigma$ to denote that the ordered pair (I, J) satisfies the constraints of $\Sigma$; furthermore, we will sometimes define schema mappings by simply defining the set of ordered pairs (I, J) that constitute M (instead of giving a set of constraints that specify M). If (I, J)∈M, we say that J is a solution of I (with respect to M).

In general, the constraints in $\Sigma$ are formulas in some logical formalism. In this paper, we will focus on schema mappings specified by source-to-target tuple-generating dependencies.

An atom is an expression of the form $R(x_1,\ldots,x_n)$. A source-to-target tuple-generating dependency (s-t tgd) is a first-order sentence of the form as follow:

$$\forall x(\varphi(x) \rightarrow \exists y\, \psi(x,y)). \tag{2}$$

where $\varphi(x)$ is a conjunction of atoms over $S$, each variable in $x$ occurs in at least one atom in $\varphi(x)$, and $\psi(x,y)$ is a conjunction of atoms over $T$ with variables in $x$ and $y$. For simplicity, we will often suppress writing the universal quantifiers $\forall x$ in the above formula. Another name for S-T tgds is global-and-local-as-view (GLAV) constraints. They contain GAV and LAV constraints, which we now define, as important special cases.

A GAV (global-as-view) constraint is an S-T tgd in which the right-hand side is a single atom with no existentially quantified variables, that is, it is denoted by the following equation:

$$\forall x(\varphi(x) \rightarrow P\,(x)). \tag{3}$$

where $P(x)$ is an atom over the target schema. A LAV (local-as-view) constraint is an s-t tgd in which the left-hand side is a single atom, that is, it is denoted by the following equation:

$$\forall x(Q(x) \rightarrow \exists y\, \psi(x,y)). \tag{4}$$

where $Q(x)$ is an atom over the source schema.

## 4 Schema Evolution Management

In this section we will discuss the details of our on-demand approach (supporting backward compatibility) and the optimization of our approach. Here, the mappings between source schema and target schema are expressed by the S-T tgds. A main requirement for database schema evolution management is thus to propagate the schema changes to the instance, i.e., to execute instance migration correctly and efficiently. S-T tgds represent the semantics of conversion from source schema instance to target schema instance. The S-T tgds can express both simple changes, such as addition, modification or deletion of individual schema constructs, and complex changes refer to multiple simple changes. The converted semantics of S-T tgds can automatically execute instance migration from source schemas to target schemas with the change of schemas.

### 4.1 Our On-Demand Approach

In our approach, the original schema and its corresponding data are not preserved. To reuse the legacy applications/queries, we must support backward compatibility. Here, virtual versions (view version) are proposed to support backward compatibility, which can avoid the costly adaptation of applications. In schema evolution progress, many times the evolution operation only involves several tables. We create views not for all the tables but only for the tables evolved.

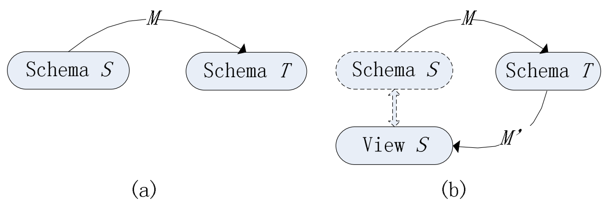

**Fig. 2.** Generation of virtual version

The functionality that our approach aims to achieve is illustrated in Fig. 2, where schema version *S* represents an initial (legacy) schema that goes through mapping *M* forming target schema version *T*. The instances migrate automatically through conversion (chase). In order to save memory space and improve performance, we delete the instances of schema version *S* and create view *S* that has the same name with the deleted schema to realize the backward compatibility. Applications deal with views the same way they deal with base tables. Supporting different explicit view versions for schemas realizes evolution transparency. The concept of view *S* can be created through mapping $M'$, which is obviously the mapping inverse of *M*.

The most important thing is to calculate the inverse of mapping *M*. The ideal goal for schema mapping inversion is to be able to recover the instances of source schema. Concretely, if we apply the mapping *M* on some source instances and then the inverse on the result of mapping *M* is used to obtain the original source instance. Here, applying a schema mapping *M* to an instance means generating the instance by chase. However, a schema mapping may drop some of the source information, and hence it is not possible to recover the same amount of information.

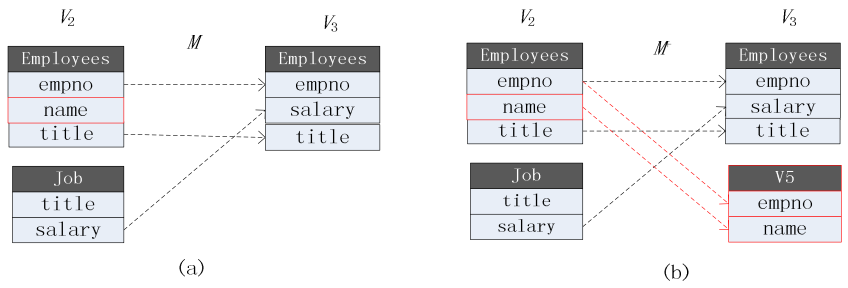

**Fig. 3.** Example of data loss

The example of the scenario described in Fig. 3(a) illustrates the schema evolution process with information loss. Consider the following two schema versions $V_2$ and $V_3$ in Fig. 1, where $V_2$ consists of two relation symbols Employees and Job, and schema $V_3$ consists of one ternary relation symbol Employees that associates each employee with salary and title. Given existing the schema mapping $M_{23}=(V_2, V_3, \Sigma_{23})$, where

$$\Sigma_{23}=\{(\text{Employees}(e,n,t)\wedge\text{Job}(t,s)\rightarrow\text{Employees}(e,s,t)\}.$$

The nature "inverse" that one would expect here is the following mapping:

$$\Sigma_{32}=\{(\text{Employees}(e,s,t)\rightarrow\exists n\text{Employees}(e,n,t),\ \text{Employees}(e,s,t)\rightarrow\text{Job}(t,s)\ \}$$

Here, we verify whether mapping $\Sigma_{23}$ can recover instances of source schema versions $V_2$. If we start with a source instance I for schema $V_2$ where the source tuples contain some constant values, and then apply the chase with mapping $\Sigma_{23}$ and then the reverse chase with $\Sigma_{32}$. Another source instance $I'$, where the tuples have nulls in the name position, is obtained. Consequently, the resulting source instance $I'$ cannot be equivalent to the original instance $I$. To give a concrete example, consider the source instance $I$ over version $V_2$ that is shown in Fig. 3.

Schema version $V_2$ could not be created from $V_3$ by mapping $\Sigma_{32}$. In our approach we also could not get the views the same as $V_2$. In order to solve this problem, a separate table is established for the lost data. In Fig. 3(b), the red rectangle represents the loss data, which are stored in table $V_3$. The special table is name $V_3$, which works only when computing mapping inverstion and evolution rollback. The special table cannot be modified. The inversion of mapping $\Sigma_{23}$ can be expressed as $\Sigma^{+}_{32}$.

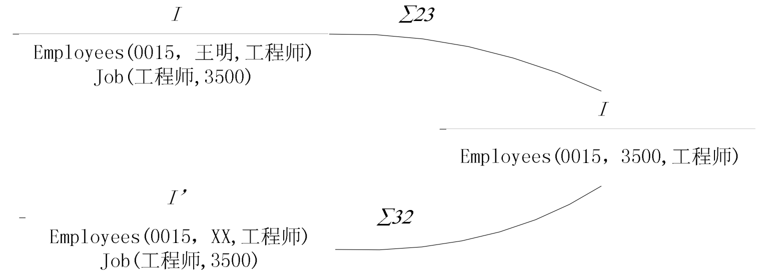
I
Σ23
Employees(0015，王明,工程师)
Job(工程师,3500)
I
Employees(0015，3500,工程师)
I'
Σ32
Employees(0015，XX,工程师)
Job(工程师,3500)

**Fig. 4.** The verification of mapping inverstion

$$\Sigma^{+}_{32}=\{((\text{Employees}(e,s,t)\vee V_5(e,n)\rightarrow\text{Employees}(e,n,t)),$$
$$(\text{Employees}(e,s,t)\rightarrow\text{Job}(t,s))\}.$$

Mapping $\Sigma^{+}_{32}$ can completely recover the instances of source schema versions. The structure of old schema versions can be correctly described by views that are created by mapping $\Sigma^{+}_{32}$. We present the algorithm for calculating the views of old schema in the following.

Algorithm 1. Virtual View Create

Input: source schema $S$, mapping $M$ between source schema and target schema

```
1:  Let $M^+=\phi$,  $M^{-1}=\phi$
2:  If $M$ is full
3:      $T$=chase$_M(I_s)$
4:      $M^{-1}$=INVERS($M$)
5:  Else
6:      $M^+$=SUPPLEMENT($M$)
7:      $T$=chase$_M{}^+(I_s)$
8:      $M^{-1}$=INVERS($M$)
9:      $V_S$=VIEW CREATE($M^{-1}$)
```

First, we determine whether there is information loss before schema evolving. If the mapping is not full, i.e., there is information loss. We will create a special table and modify the mapping $M'$. The operator of SUPPLEMENT() aims to modify the mapping $M$ without information loss. Here, data migrate is automatically executed with chase algorithm. We compute inversion of full mapping $M$ and generate the view concept with the operator VIEW CREATE().

Example of Fig. 3 is used to illustrate our algorithm. The inverse of mapping $M^+$ can be computed by operator INVERS().

$$\Sigma^+{}_{32}=\{\forall e,n,t,s(\text{Employees}(e,s,t)\vee \text{V}_5(e,n)\rightarrow \text{Employees}(e,n,t), \\ \forall e,n,t,s\ (\ \text{Employees}(e,s,t)\rightarrow \text{Job}(t,s)\}.$$

We get the mapping $M^{-1}$, which can compute the views that describe the old schema. The views concepts are computed through mapping $M^{-1}$.

```
CREATE VIEW Employees V2
        AS
    SELECT Employees.empno, V3.name, Employees.title
    FROM Employees, V3
    WHERE Employees.empno= V3.empno;
CREATE VIEW Job V2
        AS
    SELECT Employees.title, Personal.salary
    FROM Employees
```

## 4.2 Optimized Approach

In on-demand approach, we support backward compatibility to reuse the legacy applications and queries by creating virtual version. But it has scalability limitations. In fact, after several evolution steps, each application execution may involve long chain of views and thus deliver poor performance. Fig. 5 shows the limit of our naïve approach. Schema version $S_1,\ldots,S_n$ represents each version in the progress of schema evolution. $S_n$ is the current schema version, then the other schema versions are represented by views. The old schema versions connect with existing schema version through long views chains, e.g., schema version $S_1$ is mapped into schema version $S_n$ with views chain $M_1 \cup \ldots \cup M_n$. To avoid the costly implementation of applications/queries, the chains of views should get shorter as much as possible. As Fig. 5 shows, we hope that each view version would have been directly mapped into schema version $S_n$, rather than through the intermediate steps. So the implementation of applications/queries could not operate physic data through long views chains. Mapping composition is a good choice.

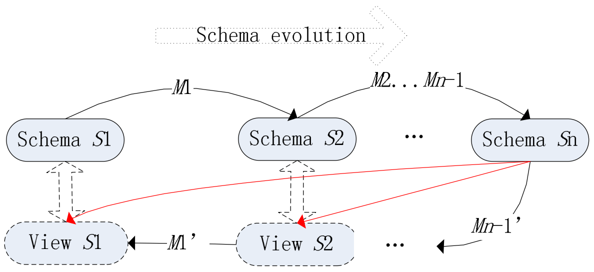

**Fig. 5.** The composition of views mapping

We do not preserve the data of old schema version. The mappings created by schema version matching are not accurate and cannot reflect the semantic of schema evolution process. In this paper, we compute the views concepts directly from existing schema version by mapping composition. The algorithm of view mapping composition is given in the following.

```
Algorithm 2. View Composition
Input: current schema version $S_c$, old schema version $S_{o1}$and $S_{o2}$, mapping $M_1$ between
$S_c$ and $S_{o1}$ and mapping $M_2$ between $S_{o1}$and $S_{o2}$
1:  Let $M^{'} = \phi$
2:  If $M_1$ is no the GAV s-t tuple tgds
3:      $M_1$ =Splitup($M_1$)
4:      $M^{'} = M_2$
5:  While the atoms of the right side of $M_1$ appear on the left side of $M_2$
6:      $M^{'}$ =REPLACE($M_1$, $M^{'}$)
7:  $M^{'}$ =SIMIPLIFY($M^{'}$)
```

Here, we illustrate the computed progress of our optimized method with an concrete example of schema evolution. We also use employees database as our example. Mapping $\Sigma_{21}$ is the relation between version $V_2$ and $V_1$.

$$\Sigma_{21} = \{(\text{Employees}(e,h,t,d) \wedge \text{Personal}(e,s,b,n) \rightarrow \text{Employees}(e,n,h,t,d)\}.$$

The view mapping from $V_3$ to $V_2$ is $\Sigma_{32}$.

$$\Sigma_{32}=\{(\text{Employees}(e,h,t,d) \rightarrow \text{Employees}(e,n,h,t,d), \text{Employees}(e,h,t,d) \rightarrow \text{Job}(t,s), \text{Personal}(e,s,b,f,l) \wedge \text{V}_5(e,n) \rightarrow \text{Personal}(e,s,b,n)\}.$$

Intuitively, the composition algorithm will replace each relation symbol from Employees and Personal in $\Sigma_{21}$ by relation symbols from that use the GAV S-T tgds of $\Sigma_{32}$. In this case, the fact Personal($e$,$s$,$b$,$f$,$l$) that occurs on the left-hand side of $\Sigma_{32}$ can be replaced by a Employees fact, according to the first GAV s-t tgd of $\Sigma_{32}$, we arrive at an intermediate tgd shown in the following.

$$\text{Employees}(e,h,t,d,s')\wedge\text{Personal}(e,s,b,n)\rightarrow\text{Employees}(e,,n,h,t,d)$$

Observe that new variable $S'$ in Employees are used instead of $S$. This avoid an otherwise unintended join with Personal, which also contains the variable $S$. We then obtain the following GLAV s-t tgd from the version $V_3$ to version $V_3$. This tgds specify the composition of $\Sigma_{32} \circ \Sigma_{21}$.

$\Sigma_{32} \circ \Sigma_{21}= \{(\text{Employees}(e,h,t,d,s')\wedge\text{Personal}(e,s,b,f,l) \wedge\text{V}_5(e,n)\rightarrow\text{Employees}(e,,n,h,t,d)$

We can get the view concept directly from schema $V_3$ through view mapping $\Sigma_{32} \circ \Sigma_{21}$.

## 5 Experimental Evaluation

In this section, we report the results of a set of experiments designed to evaluate our proposed approach for schema evolution. Table 1 describes our experimental environment. The data-set used in these experiments is obtained from the schema evolution benchmark of [17] and consists of actual queries, schema and data derived from Wikipedia. We also do some experiments on the actual database data-set.

**Table 1.**Experimental Setting

| | | |
|---|---|---|
| Machine | RAM: | 4Gb |
| | CPU(2x) | |
| | Disks: | 500G |
| OS | Distribution | Linux Ubuntu server 11 |
| MySQL | Version | 5.022 |

Now approaches for schema evolution management mostly base on version management. We get the data of wikipadia from 2009-11-3 to 2012-03-7. The size of data grows from 31.2GB to 57.8GB. If we use versioning approach to manage the schema evolution, we must spent 1056.7GB memory space to store all versions of wikipadia from 2009-11-3 to 2012-03-7. However, our approach only needs to store the existing version, i.e., 57.8GB.

We evaluate the effectiveness of our approach with the following two metrics: (1) overall percentage of queries supported, and (2) the applications/queries running time. To make comparison with the PRISM, we use the same data-set obtained from the schema evolution of [17].

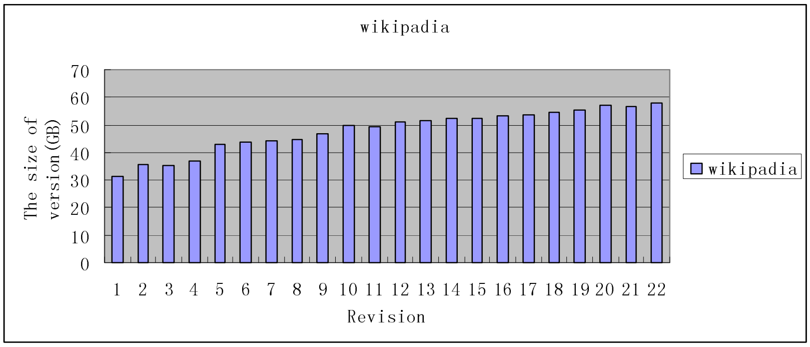

**Fig. 6.** The increase of wikipadia

**Support for Backward Compatibility.** An important measure of performance of our system is the percentage of queries supported by the old version. To this purpose we select the 66 most common query templates designed to run against version 28 of the Wikipedia schema and execute them against every subsequent schema version. The overall percentage of queries supported is computed by the following formulas.

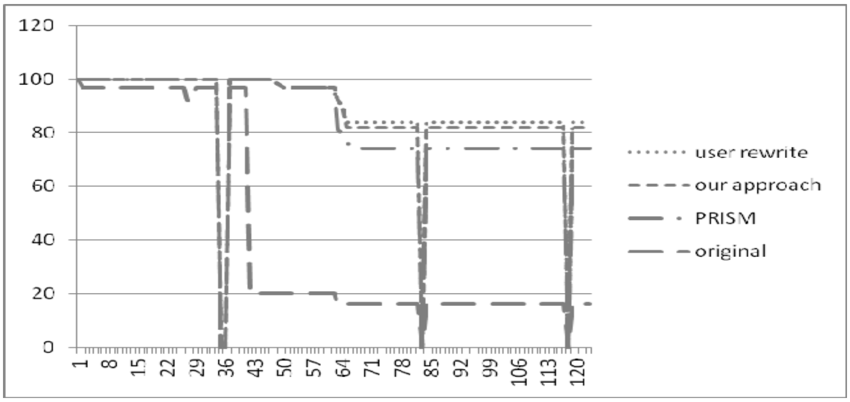

**Fig. 7.** Queries success rate

$$\text{Percent of queries supported} = \frac{\text{the queries number correctly implemented in version i}}{\text{Percent of queries supported}} \quad (5)$$

Fig.7 provides a graphical representation of the percent of queries supported for the various schema evolution approaches. Our approach can fully automatically realize the backward compatibility without rewriting queries. The original queries, failing when columns or tables are modified, highlight the portion of the schema affected by the evolution. The original query is run by the Modification. The black curve that represents the original queries shows how the schema evolution invalidates at most 82%of the schema. For PRISM, in the last version, about 26% of the queries fail due to the schema evolution. However, for our approach only 16% queries fail over the last version, which is very close to user rewritten query shown by green curve. Obviously, our approach effectively cures a wide portion of the failing input queries.

**Run-time Performance**. Here we focus on the response time of queries that represents one of the many factors determining the usability of our approach. We make a statistic for the query rewrite times of PRISM, the average of the query rewrite time for PRISM is 26.5s.

Since we cannot get the dataset of the PRIMS, we could not do the experiment to compare PRISM with our approach. Here, we use the real database data-set employees with 5 versions to compare the effectiveness of our approach with traditional view approach[10]. The database consists of approximately 500,000 tuples for about 1.3Gb of data. We selected 10 queried operated on the employees database, the response times of them are shown in Fig. 8: (1) traditional represents the approach with traditional view version; (2) on-demand is our on-demand approach; (3) optimized is our optimized approach. The testing results demonstrate that the response times of traditional approach and our naive approach grows with schema evolving, while our optimized method is close to constant. It is obvious that our optimized method has outperformed traditional approach and our naïve approach.

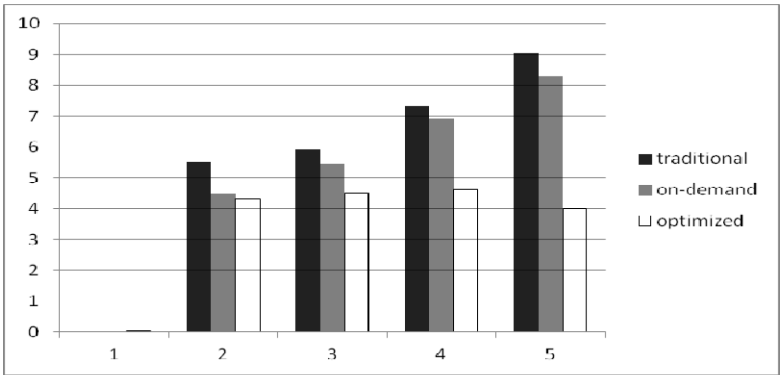

**Fig. 8.** Query execution time

## 6 Conclusion and Future Work

In this paper, we present a novel approach for schema evolution, which supports the backward compatibility. Traditionally, database schema evolution management is conducted by versioning, which generates high costs and requires much memory space. Such a time and space consuming approach severely limits the usability and convenience of the databases.

We exploit the virtual version approach which supports the applications/queries of old schemas. Our optimization for the virtual version makes our approach low time-consuming and high scalability. Both analysis and experiments verify the plausibleness of our approach and show that it consumes much less time and scales better that others schema evolution approach.

Despite recent progress we therefore see a need for substantially more research on schema evolution. For example, distributed architectures with many schemas and mappings need powerful mapping and evolution support, e.g., to propagate changes of a data source schema to merged schemas. New challenges are also posed by dynamic settings such a stream systems where the data to be analyzed may change its schema.

**Acknowledgement.** The National Natural Science Foundation of China (Grant No. 60973021, 61003060), and the Fundamental Research Funds for the Central Universities (N100704001).

## References

1. Roddick, J.F.: A Survey of Schema Versioning Issues for Database Systems. J. Information and Software Technology 37(7), 383–393 (1995)
2. Roddick, J.F.: Schema Evolution in Database System–An Annotated Bibliography. SIGMOD RECORD 21(4) (1992)
3. Banerjee, J., Kim, W.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. SIGMOD RECORD 21, 311–322 (1987)
4. McKenzie, L.E., Snodgrass, R.T.: Schema Evolution and the Relational Algebra. J. Information System 15, 195–197 (1990)
5. Liu, C.T., Chrysanthis, P.K., Chang, S.K.: Database schema evolution through the specification and maintenance of the changes on entities and relationships. In: Loucopoulos, P. (ed.) ER 1994. LNCS, vol. 881, pp. 13–16. Springer, Heidelberg (1994)
6. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G.: Schema and database Evolution in the O2 Object Dabase System. In: VLDB 1995, pp. 170–181. ACM, New York (1995)
7. Kim, W., Chou, H.-T.: Versions of Schema for Object Oriented Databases. In: VLDB 1988, pp. 148–159. Morgan Kaufmann, San Francisco (1988)
8. Munch, B.P.: Versioning in a software Engineering Database–the Changes Oriented Way. Division of Computer Systems and Telematics. Norwegian Institute of Technology (1995)
9. Andany, J., Leonard, M., Palisser, C.: Management of schema evolution in databases. In: VLDB 1991, pp. 161–170. Morgan Kaufmann, San Francisco (1991)
10. Bellahsène, Z.: View mechanism for schema evolution. In: Morrison, R., Kennedy, J. (eds.) BNCOD 1996. LNCS, vol. 1094, pp. 18–35. Springer, Heidelberg (1996)
11. Bellahsene, Z.: Schema evolution in data warehouses. J. Knowledge and Information System, 283–304 (2002)
12. Nica, A., Lee, A.J., Rundensteiner, E.A.: The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 359–373. Springer, Heidelberg (1998)
13. Rundensteiner, E.A., Lee, A.J., Nica, N.: On preserving views in evolving environments. In: KRDB 1997, pp. 13.1–13.11 (1997)
14. Karahasanovic, A., Sjøberg, D.I.K.: Visualizing Impacts of Database schema changes-A Controlled Experiment. In: HCC 2001, p. 358. IEEE, Washington, DC (2001)

15. Favre, C., Bentayeb, F., Boussaid, O.: Evolution of Data Warehouses' Optimization: A Workload Perspective. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 13–22. Springer, Heidelberg (2007)
16. Papastefanatos,G. Vassiliadis, P., Vassiliou, Y.: Adaptive Query Formulation to Handle Database Evolution (Extended Version), In: CAiSE 2006, pp.5-9, Springer, Heidelberg (2006)
17. Curino, C.A., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: the prism work-bench. In: Proc. VLDB Conf., pp. 761–772. ACM, USA (2008)