2006

A model for storage and query of cascading beliefs in multilevel security database

Jia Tao
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/880

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
A model for storage and query of cascading beliefs
in multilevel security database

by

Jia Tao

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Shashi K. Gadia, Major Professor
Leslie Miller
Douglas W. Jacobson

Iowa State University

Ames, Iowa

2006

Copyright © Jia Tao, 2006. All rights reserved.
# Table of contents

Abstract .......................................................................................................................... iii
Keywords ......................................................................................................................... iii
Contact information ...................................................................................................... iii

1. Introduction ................................................................................................................. 1
   1.1 Parametric data and parametric approach ......................................................... 8

2. Overview of the skeleton-MLS model ....................................................................... 11
   2.1 Subjects and users ............................................................................................. 11
   2.2 Objects, concurrence and belief worlds .............................................................. 11
   2.3 Path expressions ............................................................................................... 13
   2.4 Attribute values ................................................................................................ 17
   2.5 Mask: incorporating path expression in achieving persistence of algebraic identities ......................................................... 18
      2.5.1 Path expressions with the property of closure under prefixes suffice $\cup$ and $\cap$ ............................................................................ 19
      2.5.2 Difference preserves the whole skeleton with masking ............................. 20
   2.6 Relationship between a user’s view and the result of a query from this user .......... 22
   2.7 Dealing with masked path expressions .............................................................. 22
   2.8 Compatible atomic values, compatible values and equivalent terminal values .... 24
   2.9 Skeleton and tuple ............................................................................................ 27
   2.10 Dealing with masked path expressions between skeletons .............................. 31
   2.11 Compatible tuples ............................................................................................ 32
   2.12 Belief consistency among tuples ..................................................................... 32

3. Extract algorithm: generating user’s view ................................................................. 34

4. Algebra ......................................................................................................................... 38
   4.1 Domain expressions ......................................................................................... 40
   4.2 Boolean expressions ....................................................................................... 41
   4.3 Relational expressions ..................................................................................... 41

5. Update mechanism .................................................................................................... 46
   5.1 Connect clause and asserting a belief .............................................................. 46
   5.2 Disconnect clause and removing a belief ......................................................... 47
   5.3 Insert ................................................................................................................. 48
   5.4 Delete ............................................................................................................... 49
   5.5 Update .............................................................................................................. 50

6. Related works .......................................................................................................... 52

7. Conclusion and Future works .................................................................................. 54

References ..................................................................................................................... 55
Abstract

A database consists of an object space that is information about objects in (some part of) the real world. Every object existing in the real world has properties, including an identity that uniquely distinguishes it from other objects. In a belief database multiple subjects are hypothesized. Subjects have varying beliefs about existence, identities, and other properties of objects. A multilevel security database is a belief database where the subjects form a hierarchy. The upper users can see the object space of lower users but the lower users cannot. In fact lower users may not even be aware of upper users. This paper presents a skeleton based model for storage and query of a multilevel security database. This model follows the parametric approach proposed by Gadia. As in all works in the parametric approach, the central focus of this framework is to support most natural query of data.

Keywords

Multilevel Security (MLS), Database, SQL, Relational Algebra, Skeleton

Contact information

Email: jtao@cs.iastate.edu
1. Introduction

A database stores reality that is an object space consisting of information about objects in (some part of) the real world. Every object has properties, including an identity that uniquely distinguishes it from other objects. We assume that there is only one reality. An ordinary database can be used to store this single reality. However, the reality may not always be known. This is true in a belief database where multiple subjects are hypothesized, each having its own view of object space describing reality. If subjects were only interested in storage and query of their own individual beliefs, isolated ordinary databases will serve their purpose. We assume a more interesting situation that subjects are interested in query of multiple beliefs. In that case, the ordinary database paradigm is not adequate.

An interesting example of beliefs arises in the field of meteorology where scientists build numeric simulation models to predict climate. There, every simulation model (or scientist having a favorite model) is a subject and its prediction can be seen as its belief. Just as a temperature value changes from one spatial point to another, one instant to another, it also changes from one simulation model to another. Therefore, just as information can have time and space dimensions, it can also have a belief dimension. Points in belief space are nothing but subjects. As in any scientific discipline, the subjects (scientists) are not only aware of their own object spaces but also aware of object spaces belonging to other scientists. Scientific communities learn from each other for the advancement of science through mutual collaboration.

Multilevel Security (MLS) database is another example of a belief database. In MLS subjects are users who share and query information in a database. These subjects form a hierarchy induced
by a partial order. Upper subjects can see information belonging to the lower ones. On the other hand lower subjects cannot see information belonging to upper subjects. In fact the subjects may not even be aware of subjects above them. Following our assumptions, the upper subjects are interested in query of beliefs belonging to them as well as subjects below them. Every subject has its own object space. Thus subjects may differ in their beliefs about which objects exist in the real world. Obviously, in a multilevel security environment such differences are only visible to upper subjects. Even when an upper subject concurs with a lower subject on existence of an object in the real world, they may differ in their beliefs about the identity and other properties of the object. Again concurrence and lack of it on identity and other properties are also known to upper subjects only.

Example 1. Consider a hierarchy consisting of two subjects $\alpha$ and $\lambda$ where $\alpha$ is above $\lambda$. The Figure 1.1 below shows an example of their belief worlds. The objects are shown as nodes. Here objects are people in the real world and their names serve as their unique identity. The subject $\alpha$ believes that there are four objects in the real world: Shyam, Jim, Ying, and Jack, whereas subject $\lambda$ believes there are three objects in the real world: Jun, Jack, and Jackson. Note that because of the uniqueness of identity, a name cannot repeat at the same level. However, it can repeat across different levels. For example, Jack appears in object spaces of $\alpha$ as well as $\lambda$. Note that subject $\alpha$ sees its objects as well as objects belonging to $\lambda$. However, $\lambda$ only sees three objects in its object
space and not even aware of the subject $a$, let alone $a$'s object space.

It is natural to assume that the upper subject $a$ is more resourceful. It may learn that its object Shyam is known as Jun to $\lambda$ and they are not two different objects in the real world. Similarly, Jim is known as Jackson. The edges represent $a$'s concurrence with $\lambda$. Note that the concurrence is limited to existence of objects. Such concurrence applies only to identity and it does not extend to other properties of objects.

Assuming that Figure 1.1 contains the best knowledge that $a$ has, the following can be inferred by $a$. The objects Ying and Lack in its object space exist but they are unknown to $\lambda$. Also, the object known as Jack to $\lambda$ does not exist in the real world.

How would we represent such object space in a database? For example, the object Shyam belonging to $a$ that is known as Jun to $\lambda$ can be represented by the function $\{ (a, \text{Shyam}), (\lambda, \text{Jun}) \}$. Other objects are represented as $\{ (a, \text{Jim}), (\lambda, \text{Jackson}) \}$, $\{ (\alpha, \text{Ying}) \}$, $\{ (\alpha, \text{Jack}) \}$, and $\{ (\lambda, \text{Jack}) \}$. In a relational database we would represent these values under the Name column. As there are 5 objects in the belief world there would also be 5 tuples in the database. There is no loss of information as Figure 1.1 can be restored from the state of such database. Thus functions are appropriate abstraction for information in MLS. All 5 objects are visible to $a$. The information visible to $\lambda$ is a subset of this information (not a literal subset). There is no need to duplicate this information; one can simply obtain what is visible to $\lambda$ by a functional restriction to $\lambda$. For example, restriction of above objects to the domain $\{ \lambda \}$ will give $\{ (\lambda, \text{Jun}) \}$, $\{ (\lambda, \text{Jack}) \}$, and $\{ (\lambda, \text{Jackson}) \}$, precisely the objects in $\lambda$'s object space.

As far as representation of information is concerned, the ideas presented above can be
formalized to obtain a model for multilevel security. However, a difficulty arises when designing a query language. Query languages in databases are rooted in relational algebras. In other words to obtain a query language one must define relational operators first. These operators must obey good identities in order for the query language to be user-friendly. Relational algebra allows users to express their queries in different ways taking identities that equate them for granted. This principle is at the very core of databases. This is where one runs into problems in a naive approach to multilevel security.

Example 2. Consider the following relation

\[ r = \{ (\alpha, \text{Shyam}), (\lambda, \text{Jun}), (\alpha, \text{Jim}), (\lambda, \text{Jackson}), (\alpha, \text{Ying}), (\alpha, \text{Jack}), (\lambda, \text{Jack}) \}. \]

We think of a selection operator \( \sigma_\mu(r) \) to be restriction of \( r \) to domain \( \mu \). Here are some computations:

\[ \sigma_{(\alpha)}(r) = \{ (\alpha, \text{Shyam}), (\alpha, \text{Jim}), (\alpha, \text{Ying}), (\alpha, \text{Jack}) \} \]

\[ \sigma_{(\lambda)}(r) = \{ (\lambda, \text{Jun}), (\lambda, \text{Jack}) \} \}

\[ \sigma_{(\alpha) \cup (\lambda)}(r) = \{ (\alpha, \text{Shyam}), (\lambda, \text{Jun}), (\alpha, \text{Jim}), (\lambda, \text{Jackson}), (\alpha, \text{Ying}), (\alpha, \text{Jack}), (\lambda, \text{Jack}) \} = r \]

\[ \sigma_{(\alpha) \cup (\lambda)}(r) = \{ (\alpha, \text{Shyam}), (\alpha, \text{Jim}), (\alpha, \text{Ying}), (\alpha, \text{Jack}), (\lambda, \text{Jun}), (\lambda, \text{Jack}) \} \}

One would expect the identity \( \sigma_{(\alpha) \cup (\lambda)}(r) = \sigma_{(\alpha)}(r) \cup \sigma_{(\lambda)}(r) \) to hold naturally. However, this identity fails. Then how does a user know which side of the identity, if any, is correct?

[1] was Gadia’s first paper that gave a model and a query language for multilevel security. It
also showed how to solve the problem with algebraic identities discussed above. The solution required the concept of a value to be redefined in a way that it could not be destroyed by algebraic operators.

Prior to the above paper an assumption was made universally in MLS database literature that the identity of an object remains invariant among all subjects. Thus for example, the object known as Ying to $\alpha$ can only be known as Ying to $\lambda$. This assumption undermines the very concept of MLS. How would the subjects at different levels arrange identities of objects to coincide without mutual collaboration? In order to circumvent this question, a trusted agent was hypothesized who would make identities of objects available to every subject. The model [1] did not need the concept of trusted agent. The concept of trusted agent is contrary to the whole idea behind MLS.

Gadia’s second paper in MLS [4] was basically an exposition to the parametric approach to model the concept of dimension in data. It showed when identity of an object is required to remain the same at all levels, why there is hardly any difference between a temporal database model and a model for MLS. In the former the points in the parametric space are instants of time and in the latter they are subjects. The query languages for the two databases in the parametric approach have the same grammatical structure. The difference between them is rather small in that the linear structure on time and a partial order on subjects simply give rise to different atomic constructs. The paper agrees that the generic ParaSQL obtained in this way is more natural for users in general, as [10] in particular, than all models in MLS that existed at the time.

As stated above, [1] was the first paper that allowed subjects the independence to have varying identities to objects. Subsequently, some papers have been published, such as [8], that also allow
identity of an object to vary. But these models are difficult for users when compared to [1]. The parametric approach is completely user-centric in breadth as well as depth. The philosophy of the parametric approach is that a user should be able to take for granted constructs that are natural in a certain context. This forces the complexity of modeling to be absorbed by the system rather than passing it on to the user.

In this paper we take our model of MLS one step further. The single focus of our work is to continue to produce a query framework that is most natural for users. This has to be done without compromising the integrity of information as alluded in Example 2. In addition, we feel that our framework will support efficient query processing. The queries that we can express naturally, will be very difficult to express in other models. Once the linguistics constructs in our query language are understood they can in fact augment natural languages to help MLS queries to be expressed more easily. The following example gives a glimpse of the issues we are trying to address in the new model of multilevel security.

**Example 3.** Consider a three subject hierarchy consisting of subjects $\gamma$, $\alpha$, and $\lambda$. where $\gamma$ is the highest subject, $\lambda$ is the lowest and $\alpha$ is between the two. Object spaces of these subjects are shown in Figure 1.2.

![Figure 1.2. An example of object spaces from different users’ views](image-url)
We draw attention to the object Jack belonging to subject $y$ from its own point of view. According to $y$, this object is known as Jack to $\lambda$ and as Jim to $a$. However from $a$’s point of view the object is known as Jackson to $\lambda$. This entire observation could be an answer to the query that asks for all cascading identities of the object Jack belonging to subject $y$. In parametric model we regard this query as important and make it as easy for a user to express as possible. The problem is that this information is no longer a simple function because at $\lambda$ we have Jack as well as Jackson giving rise to the pairs $(\lambda, \text{Jack})$ and $(\lambda, \text{Jackson})$ that cannot belong to the same function.

This problem can be solved by partitioning the above hierarchy into two levels at a time, and that is precisely what was done in [1], Gadia’s first work in MLS. But the problem with that is from a user’s point of view the query will require complex join operations. Furthermore, what is a join in this three level hierarchy will become multi-way join when the subject hierarchy is deeper. Thus the grammatical structure of the query would depend upon the subject hierarchy. This can be jarring to a user. For example, a query that worked in past may not work any longer if a subject is added at a lower level. Furthermore, it turns out that queries such as this can be complex even in a natural language, although one has a better hope of expressing it independently of object hierarchy. The query language we present will have phrases that could be seen to augment the natural language itself. An extended natural language, obtained in this way, would be much better equipped to pause useful queries in multilevel security.

We also observe that in our examples considered so far we only have used subject hierarchies that were linear, where every subject is either below or above another subject. In [1] we considered the subjects to be partially ordered. Thus subjects could form a lattice shown in Figure
1.3. A lattice is not necessary, however. For example if either γ or λ is not present we do not have a lattice but the remaining subjects are still partially ordered. We also note that if γ as well as λ were not present, the subject space would reduce to two discrete levels α and β having no relationship with each other. Although our model would handle it, in such a case one would obtain a disjoint union of two ordinary databases. That is not an interesting scenario. In other words multilevel security is interesting only when there is a hierarchy. Nonlinear hierarchies have been alluded to in some early papers in multilevel security; however [1] seems to have given the first model and a query language where a non-linear hierarchy was allowed.

![Figure 1.3. Example of partial order of user levels in MultiLevel Security Databases](image)

**Figure 1.3. Example of partial order of user levels in MultiLevel Security Databases**

### 1.1 Parametric data and parametric approach

The term *parametric data* applies to all data where the values are tied down to points in some space that is simply termed the *parametric space*. Examples of parametric spaces are set of instants in temporal data, spatial points in spatial data, and subjects in belief data of which multilevel security is a special case. Thus all works in multilevel security databases, including [9], as well as those in temporal and spatial databases are works in parametric databases.

A parametric space may be multidimensional. When the parametric space is restricted to a single point, a value becomes fixed. If this parametric point is ignored, the parametric database reduces to an ordinary (classical) database. In other words, a parametric database restricted to a
single point in the parametric space is isomorphic to an ordinary classical database. Therefore, a
classical database is a legitimate, albeit a simple and degenerate, case of parametric database. An
ordinary database can be considered a parametric database where the underlying set of
dimensions is empty. All concepts arising in ordinary classical databases (must and do) embed in
parametric databases; this is not a matter of choice.

The term *parametric approach* is used when the association of a value to points in the
underlying parametric space is explicitly recognized and leveraged to study parametric databases.
This brings many aspects of parametric data to surface and leads to a clearer understanding of
parametric data. For example, consider the issue of nulls in databases. Often, in works in classical
databases one assumes that there are no nulls in order to develop a core model and relational
algebra, which in turn lead to user oriented query languages such as SQL. It is well known that
the presence of nulls significantly complicate a query language. Parametric approach does not
profess that one should not deal with nulls. However, it is important to be aware of nulls when
they are present.

What is the counterpart of non-nulls in parametric data? A tuple in a parametric approach is
said to be *homogeneous* if all attribute values in the tuple have the same parametric domain. In
absence of homogeneity, the database is bound to have nulls when restricted to single point. The
homogeneity assumption allows us to obtain a core model, algebra, and SQL-style query language
for parametric data much in the same way that the same exercise is carried on in classical model
without nulls prior to taking nulls into consideration.

Parametric approach also offers advantages. For example, any generic concept developed for
one type of dimension applies readily to other dimensions. As discussed above, [4] argues that a framework for temporal databases applies readily to multilevel security when all users are required to agree on key-values of an object. Another advantage of the parametric approach is that it makes a clear distinction between a \textit{weak concept} and a \textit{strong concept}. This classification stems from the two notions of equality among relations: weak equality and strong equality. ("Strong" is used for emphasis and to contrast it from "weak"). Weak equality only requires two relations to have same information at every point in the parametric space. It ignores how information at different points is combined together. (Strong) equality requires the two relations to be identical. All concepts in parametric databases can be classified as weak or strong. For example the identity $r \cup s = s \cup r$ is a weak identity and not a strong identity simply because $r(p) \cup s(p) = s(p) \cup r(p)$ holds at every point $p$ in the parametric space. An implication of this is that this identity is not new in parametric databases. Weak concepts are simply counterpart of classical concepts, whereas strong concepts are truly new. Parametric approach also does not subscribe to any fixed paradigm, e.g. relational, object-oriented, or XML. However, it imposes a very strict discipline on how the concept of dimension is added.
2. Overview of the skeleton-MLS model

2.1 Subjects and users

In multilevel security databases, multiple users with a partial order $(\leq)$ among them form a security hierarchy. We call these users subjects [6]. For simplicity, we assume that in each user level, we only use one user. In other words, we don’t distinguish users at the same level. In previous section, a need was alluded to for modeling and query of all cascading beliefs relative to a subject. Informally, a skeleton assembles such cascading beliefs. Skeletons are formalized as values in the parametric model. Then one goes on to derive tuples and relations, paving a foundation to algebraic operators, which would in turn lead to SQL-style query language.

We will use “user”, “user level” and “subject” interchangeably. We denote the set of all users as $U$. For two users $u_1$ and $u_2$, if $u_1 \leq u_2$ and $u_1 \neq u_2$, then $u_1$ is said to be below $u_2$ or $u_2$ is above $u_1$. Recall in Figure 1.3, we have an example with a partial order $\alpha \leq \gamma$, $\beta \leq \gamma$, $\lambda \leq \alpha$, $\lambda \leq \beta$, and $\lambda \leq \gamma$ which is derived from transitivity. Bell Lapadula model tells us that an upper level user has read access to lower level user and that a lower level user has write access to upper level user in MultiLevel Security Database [6].

2.2 Objects, concurrence and belief worlds

We assume there is only one reality of data in the real world. The real world consists of a set of objects, called the object space. However, different users have varying views of object space. Each object, according to a user’s view, has a set of properties, which are also known as attributes. We use identity of each object as the key attribute. Thus, objects are uniquely identified by the
key attribute. As an example in Figure 1.2, assuming the key attribute is name, in user γ’s object space, there are exactly three objects: Jack, Hari and Lan. User α’s object space is {Shyam, Jim, Ying, Jack}. And user λ’s object space is {Jun, Jack, Jackson}. As an extension of example in Figure 1.2, Figure 2.2.1 lists object spaces with attribute values for each user in Figure 1.3.

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jack</td>
<td>100K</td>
<td>Toys</td>
</tr>
<tr>
<td>Lan</td>
<td>65K</td>
<td>Shoes</td>
</tr>
<tr>
<td>Hari</td>
<td>50K</td>
<td>Toys</td>
</tr>
<tr>
<td>α</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jack</td>
<td>40K</td>
<td>Shoes</td>
</tr>
<tr>
<td>Shyam</td>
<td>50K</td>
<td>Toys</td>
</tr>
<tr>
<td>Jim</td>
<td>60K</td>
<td>Toys</td>
</tr>
<tr>
<td>Ying</td>
<td>65K</td>
<td>Toys</td>
</tr>
<tr>
<td>β</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jose</td>
<td>40K</td>
<td>Toys</td>
</tr>
<tr>
<td>Olga</td>
<td>50K</td>
<td>PCs</td>
</tr>
<tr>
<td>Jackson</td>
<td>60K</td>
<td>Clothes</td>
</tr>
<tr>
<td>γ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jack</td>
<td>60K</td>
<td>Clothes</td>
</tr>
<tr>
<td>Jun</td>
<td>70K</td>
<td>PCs</td>
</tr>
<tr>
<td>Jackson</td>
<td>80K</td>
<td>Toys</td>
</tr>
</tbody>
</table>

Figure 2.2.1. Object spaces

Recall that an upper subject has read access to lower subject and that a lower subject has write access to upper subject in Multilevel Security Databases. This property is also represented in the Figure 1.2. The edges show the concurrences to objects from an upper user to a lower user. For example, the edge from (γ, Jack) to (α, Jim) implies one of γ’s concurrences with α, i.e., the object known as Jim to α is known as Jack to γ; the edge from (γ, Jack) to (λ, Jack) tells us Jack at λ is known as Jack to γ, which is another concurrence of γ; the edge from (α, Jim) to (λ, Jackson) is a concurrence of α, showing that in α’s point of view, Jackson at λ is the same object as Jim at α.

Due to the existence of edges from (γ, Jack) to (α, Jim) and from (α, Jim) to (λ, Jackson), we cannot have another edge from (γ, Jack) to (α, Ying) or from (α, Jack) to (λ, Jackson), which are
shown by using dashed lines in Figure 1.2. The existence of such dashed lines is meaningless because it implies the confliction of the upper user’s belief.

For those objects that don’t have any edge connected from an upper level, they are not believed to exist in the real world by that specific upper level user. For example, in Figure 1.2, Jack at α level is believed to be a fake object by γ; α doesn’t think Jack at λ exists in real world.

A subject has interest in its own object space, object space belonging to lower users as well as cascading concurrences about identities of objects. Collectively such object spaces and concurrences are called the belief world of a subject. For example, in Figure 1.1, the belief world of α could be represented as {{(α, Shyam), (λ, Jun)}, {{α, Jim}, (λ, Jackson)}, {{(α, Ying)}, {{α, Jack}}, {(λ, Jack)}}.

2.3 Path expressions

A parametric space of points is hypothesized in the parametric approach to databases. Subjects can be treated as points in parametric space in multilevel security databases. A value is a function from a subset of the parametric space. And this subset is the domain of the function.

We want to preserve user’s cascading view of an object into one tuple so that we can design an efficient query language for users to ask questions about multiple beliefs. If we save user’s cascading view of an object, i.e., objects at different levels but are believed directly or indirectly as one object by users, into one tuple instead of breaking them each into one tuple, join will be largely reduced. As an example of Figure 2.3.1, (a) is a graph representation of γ’s cascading view of Jack; (b) is a tabular representation by using subjects as parametric points.
We can see that using only subjects as points, as in Figure 2.3.1 (b), to represent the subject’s cascading view of an object cannot form a function. There are two values at \( λ \) level (\( λ \), Jack) and (\( λ \), Jackson). We cannot put them into one value. Furthermore, we will lose all concurrences between different user levels. We are not able to find out whether (\( λ \), Jackson) is connected with (\( α \), Jim) or (\( β \), Jose). When we have a graph representation, we can get this tabular representation. However, with this tabular representation, we cannot restore the graph. Besides these, as we mentioned in Example 2, algebraic identities will break down by using subjects themselves as parametric points.

To solve the above problems, we use path expressions as parametric points. A path expression
is a container that is used to represent a higher user's concurrence with a lower user. If \( u_1 \) is above \( u_2 \), then \( u_1.u_2 \) is a path expression that is used to represent \( u_1 \)'s concurrence with \( u_2 \). Only \( u_1 \) knows this concurrence, \( u_2 \) doesn't know anything about it since \( u_2 \) isn't even aware of the existence of \( u_1 \) because of the partial order.

We can also recursively define path expression. The path expression \( u_1.u_2.u_3 \ldots u_n \) means each user is in concurrence with the user right after him, and whatever next user believes, he doesn't care. For example, \( u_1.u_2.u_3 \) means \( u_1 \) is in concurrence with \( u_2 \) and whoever \( u_2 \) is in concurrence with, \( u_1 \) doesn't care. This helps us recognize the difference between \( \gamma.\lambda \) and \( \gamma.a.\lambda \). Whereas \( \gamma.\lambda \) represents concurrence of \( \gamma \) with \( \lambda \), \( \gamma.a.\lambda \) does not. The latter represents \( \gamma \)'s concurrence with \( \lambda \) via \( \alpha \), i.e., \( \gamma \) concurs what \( \alpha \) believes and \( \alpha \) concurs what \( \lambda \) believes.

Figure 2.3.1 (c) is a tabular representation of \( \gamma \)'s cascading view of Jack by using path expressions as parametric points. Every path expression in the tuple appears no more than once and is closed under prefixes. The situation that one atomic value is mapped from two path expressions might happen, but no two values are mapped from one path expression. With this representation, we keep all the cascading beliefs, from different levels related to an object, in one tuple.

With the property of path expressions closed under prefixes in a tuple, concurrences are easily inferred. As an example in Figure 2.3.1 (c), we are able to figure out that Jim known to \( \alpha \) is known as Jackson to \( \lambda \). Each user's belief is persistent and traceable when incorporating path expression and the attribute value. We don't save the information about the edges appeared in Figure 2.3.1 (a), which represent concurrences, since all the concurrence information can be
easily implied from path expressions within a tuple. Edges are actually virtual. We only keep such
tree-like structure (with edges) appearing in the paper for good visualization and easy understanding.

Let’s give a scenario with concurrences for the objects in Figure 2.2.1:

![Diagram of belief worlds](image)

**Figure 2.3.2. Example of Emp relation: scenario with concurrences**

Figure 2.3.2 (a) shows γ’s belief world. Figure 2.3.2 (b) is α’s belief world, which could be
generated from Figure 2.3.2 (a).

With the partial order of user levels we mentioned in Figure 1.3, the set of all possibilities of
valid path expressions representing concurrences is \{γ, α, β, λ, γα, γβ, γλ, αλ, βλ, γαλ, γβλ\}.
We call this set the set C, meaning the set of concurrence. Each element in set C is a parametric
element.
From now on, we have a new definition of the parametric space in our model. Every element in set C is a point in the parametric space. For each point in C which has only one user level, it represents the user’s belief to an object at its own object space; for elements which have more than one user levels, the last user level tells us in which level this object belongs to, and the prefix forms a path of the belief.

Example 4. Suppose we want to represent Jackson at λ level from γ’s point of view in Figure 2.3.2, by using path expression as domain of function to attribute NAME, we will have (γ.a.λ, Jackson). With this representation, we can easily infer:

1. Jackson is an object belonging to user λ.

2. This object is in user γ’s belief world as well as α’s belief world. However, this doesn’t imply the object from path expression α.λ is only in α’s belief world. It’s also in γ’s belief world, yet, not in γ’s object space.

3. Jackson is known as some object, let’s say a, at α level. Furthermore, a at α level is known to be some object, let’s say b, in γ’s object space.

2.4 Attribute values

A C-assignment to an attribute A of an object is a function \( \xi \) from a parametric element into \( \text{dom}(A) \), the domain of A. For example, according to Figure 2.3.2 (a), \{γ, Hari \}, (γ, Jack), (γ, Lan)\} is a γ-assignment to the attribute NAME; \{(a.λ, Jun), (a.λ, Jackson)\} is a a.λ-assignment to the attribute NAME.

The domain of a security value \( \xi \) is denoted as \([\xi]\). For example, \([\{(γ, Jack), (γ.a, Jim), (γ.a.λ, Jackson)\}] = \{γ, γ.a, γ.a.λ\}. We assume that the attribute values within the same tuple have the
same domain. This assumption, termed homogeneity assumption in the parametric model, is the counterpart of classical database without nulls.

2.5 Mask: incorporating path expression in achieving persistence of algebraic identities

In MLS, the most interesting queries are queries about different beliefs. As an example, user $\gamma$ might ask “give me all the objects in my belief world which are known as Jack (directly or indirectly) in my object space”. Our goal is to provide a query language for users to make such queries as easy as possible.

![Diagram](image)

Figure 2.5.1. Masking helps result maintain one value

Every user should have full knowledge of all the concurrences and objects in its belief world. Whenever a query happens, the result might chop cascading beliefs of an object into several pieces. Some pieces belong to the user’s belief world; however, they are not originally queried.

For example, Figure 2.5.1 (a) is $\gamma$’s cascading view of object Jack. It is one value in our model. What would it be if we just select two objects, Jose at $\beta$ and Jackson at $\lambda$, out of this value? We still want to keep these two objects in one value so that the result is consistent with the original value. To fulfill this purpose, we have to keep Jack at $\gamma$ and Jim at $\alpha$ as well in the query result. To
differentiate the wanted query result and these auxiliary information, we mask Jack at $\gamma$ and Jim at $\alpha$ so that the user knows it’s not what he originally queried about. The user who did the query can see the masked part; however, he knows this part as an auxiliary in the query result. The mask is denoted by using a separator $||$ after the domain, representing the point is masked. If the domain of an object is masked, the object itself is also masked.

With mask, a path expression can be represented as $p||$, where $p$ is a legal path expression, i.e., a legal parametric element. If a path expression is masked, the attribute value from this path expression is also masked.

### 2.5.1 Path expressions with the property of closure under prefixes suffice $\cup$ and $\cap$.

![Figure 2.5.2. Path expressions with closure under prefixes suffice $\cup$ and $\cap$](image)
Figure 2.5.2 is an example of how to perform union of two selections. In order to make identity that selection of union equals to union of selections hold, in every selection, for each object we query about, we will keep all the objects from root along the belief path until this object, with objects that we didn’t query about being masked. Note that the result in Figure 2.5.2(d) is same as Figure 2.5.1(b).

By doing this, the path expressions in every value (no matter it is an original value in database or generated after some queries) are closed under prefixes. This property of path expressions in a value always gives us a path of beliefs from the root till any object in this value.

Not only does the operator of union need this property, intersection also requires path expressions in a value closed under prefixes.

2.5.2 Difference preserves the whole skeleton with masking.

With the example showed in Figure 2.5.2, we know that path expressions with closure under prefixes suffice union and intersection. However, this property only fails the difference operator. Consider the example showed in Figure 2.5.2, how can we perform $S_1 - S_2$? According to parametric approach, we will first consider $\text{dom}(S_1) - \text{dom}(S_2)$. When we take a look at each pair path expressions of the atomic values, we could get the combination of masked path expression, unmasked path expression as well as null. How could we define the operation of difference between any two of them? For example, path expressions of Jack at $\gamma$ in both $S_1$ and $S_2$ are masked. Should the difference between these two path expressions be null? This is obviously wrong since we need to keep path expressions closed under the prefixes. This implies that we shouldn’t have null atomic values after difference operation as long as some atomic value(s) is not
null within the same value. Therefore, the way how S1 and S2 form is questioned. For S1, all the
unneeded tailings were deleted, resulting nulls in atomic values. Same applies to S2. If we should
keep every atomic value, those masked tailings would be kept. Then with this domain, we
compute the function. This results in keeping the whole skeleton with some part being masked
and others not.

![Diagram](image)

**Figure 2.5.3 Difference preserves whole skeleton with masking**

Figure 2.5.3 gives us the same queries S1 and S2, except that we keep the whole skeleton, with
irrelevant objects masked and wanted objects unmasked. Now we know how to perform
difference between these two values. A formal definition about algebraic operations of
masked/unmasked path expressions is defined in 2.8.

An important property of mask is that the path expression without a mask always dominates the
same path expression with a mask. For example, \{p||\} ⊔ \{p\} = \{p\}. This property is used in the
example given in Figure 2.5.3. More examples of algebraic operations will be given in section
2.8.
2.6 Relationship between a user’s view and the result of a query from this user

We’ve talked about a user’s view sometimes. A *user’s view* is the user’s belief world, i.e., all that the user can see. It includes all the objects belonging to this user, all the objects of lower user(s), and all the cascading concurrences.

The result of a query asked by this user is a subset of this user’s view. In order to make the result persistent to the original view and the algebraic identities to hold, the whole skeleton will be kept in the result and mask will be applied if necessary.

2.7 Dealing with masked path expressions

Two path expressions $p_1$ and $p_2$ are said to be *same* if $p_1=p_2$. For example, none of the path expressions $\gamma.a$, $\gamma.\lambda$, $\gamma.a|$ is same as the other.

Two path expressions $p_1$ and $p_2$ are said to be *similar*, denoted as $p_1\sim p_2$, if either they are same or they differ only in their masks. For example, $a.\lambda\sim a.\lambda|$.  

If $p$ is a path expression without mask, $p|$ denotes the similar path expression with mask. If $p$ is a path expression with or without mask, $[p]$ denotes a path expression similar to $p$, but without mask. If $P$ is a set of path expressions, $[P]$ is a set formed by similar path expressions without mask of all the path expressions in $P$.
**Definition 1:** Suppose $x$ and $y$ are two similar path expressions, and $p$ denotes a path expression without mask. The computation of union, intersection, difference and complementation of $x$ and $y$ is given below. Note that this table is isomorphic to truth table, where $p$ behaves as TRUE and $p||$ behaves as FALSE.

**Definition 2:** Suppose $C$ is the set of parametric space, $P_1$ and $P_2$ are two sets of path expressions with elements possibly masked. In the following, the computation of union, intersection, and difference of $P_1$ and $P_2$ is defined by using the literal union, intersection, and difference. In each equation, the symbols $\cup$, $\cap$, $-$ on the right hand side of equal sign denote literal union, literal intersection and literal difference respectively.

\[
P_1 \cup P_2 = \{ x : [x] \in [P_1] - [P_2] \} \bigcup \{ x : [x] \in [P_2] - [P_1] \} \bigcup \{ x \wedge y : x \in P_1, \text{ and } y \in P_2, \text{ and } x \sim y \}
\]

\[
P_1 \cap P_2 = \{ x \wedge y : x \in P_1, \text{ and } y \in P_2, \text{ and } x \sim y \}
\]

\[
P_1 - P_2 = \{ x : [x] \in [P_1] - [P_2] \} \cup \{ x - y : x \in P_1, \text{ and } y \in P_2, \text{ and } x \sim y \}
\]

Since in our model, we will never use $\cup$, $\cap$, $-$ to denote literal union, intersection and difference later on, we will use "$P_1 \cup P_2$" to denote "$P_1$ union $P_2$", "$P_1 \cap P_2$" to denote "$P_1$ intersection $P_2$", and "$P_1 - P_2$" to denote "$P_1$ difference $P_2$" from now on.

**Example 5.** Suppose $P_1 = \{ \gamma, \gamma.a, \gamma.\lambda, \gamma.a.\lambda || \}$, $P_2 = \{ \gamma||, \gamma.a||, \gamma.\lambda||, \gamma.a.\lambda\}$, then $P_1 \cup P_2 = \{ \gamma, \gamma.a, \gamma.\lambda, \gamma.a.\lambda, \gamma.a.\lambda || \}$, $P_1 \cap P_2 = \{ \gamma||, \gamma.a||, \gamma.\lambda||, \gamma.a.\lambda|| \}$, $P_1 - P_2 = \{ \gamma, \gamma.a, \gamma.\lambda, \gamma.a.\lambda\}$, $P_2 - P_1 = \{ \gamma||, \gamma.a||, \gamma.\lambda||, \gamma.a.\lambda\}$. 
2.8 Compatible atomic values, compatible values and equivalent terminal values

With the concept of masked path expressions, we say two atomic values at point $p_1$ and $p_2$ respectively (i.e., two objects) are compatible if $V_1(p_1)=V_2(p_2)$ and $p_1 \neq p_2$. In another word, when two atomic values are compatible, they are same object with similar path expressions.

**Definition 3:** Suppose two atomic values $V_1(p_1) = (p_1, a)$ and $V_2(p_2) = (p_2, a)$ are compatible, then the union, intersection, and difference of these two compatible atomic values are defined as following:

$$V_1(p_1) \cup V_2(p_2) = (p_1 \lor p_2, a)$$
$$V_1(p_1) \cap V_2(p_2) = (p_1 \land p_2, a)$$
$$V_1(p_1) - V_2(p_2) = (p_1 - p_2, a)$$

**Example 6.** Algebraic operations for compatible atomic values.

$$(\gamma, \text{Jim}) \cap (\gamma||, \text{Jim}) = (\gamma||, \text{Jim})$$
$$(\gamma, \text{Jim}) \cup (\gamma||, \text{Jim}) = (\gamma, \text{Jim})$$
$$(\gamma, \text{Jim}) - (\gamma||, \text{Jim}) = (\gamma, \text{Jim})$$

Two values $V_1$ and $V_2$ are said to be compatible if they are rooted in compatible atomic values.

Figure 2.8.1 gives us three compatible values. (a), (b), (c) are tabular representations and (d), (e), (f) are corresponding graph representations by circling masked objects.

Union, intersection and difference of two compatible values result in one value. Their roots help us achieve persistence of algebraic identities when performing union and/or intersection.
In section 2.5, we already mention that path expressions with the property of closure under prefixes only already suffice union and intersection. Readers might already notice that in Figure 2.5.2, except the wanted objects and objects from root along the belief path are preserved, all the rest objects are deleted. However, the operator difference preserves the whole skeleton.

\[
\begin{array}{c}
\gamma, \text{ Jack} \\
\gamma.\alpha, \text{ Jim} \\
\gamma.\beta, \text{ Jose} \\
\gamma.\lambda, \text{ Jack} \\
\gamma.\beta.\lambda, \text{ Jack} \\
\gamma.\alpha.\lambda, \text{ Jackson}
\end{array}
\]

\[
\begin{array}{c}
\gamma, \text{ Jack} \\
\gamma.\alpha, \text{ Jim} \\
\gamma.\beta, \text{ Jose} \\
\gamma.\lambda, \text{ Jack} \\
\gamma.\beta.\lambda, \text{ Jack} \\
\gamma.\alpha.\lambda, \text{ Jackson}
\end{array}
\]

\[
\begin{array}{c}
\gamma, \text{ Jack} \\
\gamma.\alpha, \text{ Jim} \\
\gamma.\beta, \text{ Jose} \\
\gamma.\lambda, \text{ Jack} \\
\gamma.\beta.\lambda, \text{ Jack} \\
\gamma.\alpha.\lambda, \text{ Jackson}
\end{array}
\]

\[
\begin{array}{c}
\gamma, \text{ Jack} \\
\gamma.\alpha, \text{ Jim} \\
\gamma.\beta, \text{ Jose} \\
\gamma.\lambda, \text{ Jack} \\
\gamma.\beta.\lambda, \text{ Jack} \\
\gamma.\alpha.\lambda, \text{ Jackson}
\end{array}
\]

Figure 2.8.1. Compatible values

To deal with union, intersection and difference uniformly, during the computation, we always keep the whole skeleton with some objects masked when necessary. After we finish computation, we can prune the masked tails in a skeleton.

Suppose \( V_1 \) and \( V_2 \) are two values, we say \( V_1 \) and \( V_2 \) have equivalent values, \( V_1 \equiv V_2 \), if for any unmasked object in \( V_1 \), there is also a same unmasked object in \( V_2 \), and all the objects from the roots along the path to these two unmasked objects in \( V_1 \) and \( V_2 \) respectively are pairwise same.

The word “same” implies the same objects with same path expressions.
Suppose we have an employee relation shown in Figure 2.3.2, and γ wants to know what is in user α’s belief about Jim in its own object space, Figure 2.8.2 answers the question by giving us a group of equivalent terminal values which are in graph representation. All the masked objects are circled. Each of these equivalent values can be retrieved as the result of this query. Among them, (a) has the most complete information of this value. It contains the whole skeleton. We call this value \( \text{max}(V) \). However, some information in \( \text{max}(V) \) is totally irrelevant, such as Jose in β level, since α and β are two independent users. (d) has the least but complete information that are necessary. We call it an equivalent terminal value of these equivalent values.

![Diagram](image1)

(a) \( V_1 \)  
(b) \( V_2 \)  
(c) \( V_3 \)  
(d) \( V_4 \)

**Figure 2.8.2. Equivalent values**

Within an equivalent terminal value, all the path expressions satisfy the property of closure
under prefixes. It maintains a full path of beliefs for any unmasked objects, starting from the root object to this unmasked object. After any algebraic computation, the value we get is $\text{max}(V)$. For simplicity, we will use equivalent terminal value instead to show the results of queries later on.

2.9 Skeleton and tuple

We already know that if we want to represent an object of the highest user’s cascading view among several user levels, we use path expressions as points in domain with the property of closure under prefixes. Each object from one path expression is a belief of some user. All these objects as well as the concurrences among them form a rooted tree-like graph. In Figure 2.3.2 (a), we have seven such kind of rooted “trees” (each object without any concurrence by itself is a tree, and this kind of trees only has a root). We call each such kind of rooted tree a skeleton.

Each skeleton represents a relationship between a set of objects in different levels from different users’ view. These objects are either directly related to each other or indirectly related to each other. All the pairs of two objects in one skeleton are rooted in the same root, i.e., the roots of any pair of objects have compatible atomic values. For example, $(y, \text{Jack})$ and $(y||, \text{Jack})$ are same root, but $(y, \text{Jack})$ and $(\lambda, \text{Jack})$ are different roots.

In the context of parametric approach, each valid path expression is a point in parametric space. When mapping them to a skeleton, each path expression is also a point sitting in the last user level of this path expression and showing the path of belief. For example, $(\gamma.a.\lambda, \text{Jackson})$ in Figure 2.5.1 gives us a parametric point $\gamma.a.\lambda$. The value of this point is Jackson and this point resides at $\lambda$ level. Incorporating the property of closure under prefixes, we can easily figure out the path of belief. In Figure 2.5.1, $\{(\gamma, \text{Jack}), (\gamma.a, \text{Jim}), (\gamma.\beta, \text{Jose}), (\gamma.a.\lambda, \text{Jackson})\}$ gives us a hint that one
path of belief is from user $\gamma$ to user $\alpha$, and then extend to user $\lambda$. Jack known to $\gamma$ is known as Jim to $\alpha$; Jackson known to $\lambda$ is known as Jim to $\alpha$. When we look at the object $(\gamma, \alpha, \lambda, \text{Jackson})$, we can also trace the belief by tracing the prefixes of this path expression in the same tuple.

Recall that we used edges between two parametric points (path expressions) to explain concurrences. Now we can easily figure out the concurrences in a tuple by using the property of closure under prefixes of path expressions in a tuple. We don’t save those edges in our model. However, we keep them in skeleton graph for users to understand the idea easily.

When a query is performed, we might only want an object from lower user level instead of the whole cascading view from an upper user. Thinking of a skeleton, if we just withdraw the needed point from the whole tree, we will lose all the concurrence information related to this object. Then the result of this query is not consistent and complete.

When we say consistent, we mean the result of a query should be consistent with the original database and it should be within a certain user’s view (depending on who is asking the query); when we say complete, we mean we should preserve the integrity of the data. We shouldn’t lose any necessary information.

Our solution, the skeleton-based MLS model, will keep the root as well as the path information in any query result. Since this information is valid for the query-asking user to know, it is legal to appear in the query result. However, to differentiate the wanted result and the auxiliary information, we mask all the objects which are not originally wanted in the result by using separator || after the path expression(s). In Figure 2.5.1, the root was masked. Each tuple in the result of such a query should have the full path from root till the object which was queried,
incorporating with mask if necessary. As a result, the domain of each tuple is closed under prefixes.

So far, we only talk about the identities of objects, specifically, “Name” in this context. What about the other properties of an object? A tuple in a parametric approach is said to be **homogeneous** if all attribute values in the tuple have the same parametric domain. In absence of homogeneity, the database is bound to have nulls when restricted to single point. The homogeneity assumption allows us to obtain a core model, algebra, and SQL-style query language for parametric data much in the same way that the same exercise is carried on in classical model without nulls prior to taking nulls into consideration.

![Skeleton and tuple](image)

(a) Attribute values

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ, Jack</td>
<td>γ, 100K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td>γ.a, Jim</td>
<td>γ.a, 60K</td>
<td>γ.a, Toys</td>
</tr>
<tr>
<td>γ.β, Jose</td>
<td>γ.β, 40K</td>
<td>γ.β, Toys</td>
</tr>
<tr>
<td>γ.λ, Jack</td>
<td>γ.λ, 60K</td>
<td>γ.λ, Clothes</td>
</tr>
<tr>
<td>γ.β.λ, Jack</td>
<td>γ.β.λ, 60K</td>
<td>γ.β.λ, Clothes</td>
</tr>
<tr>
<td>γ.a.λ, Jackson</td>
<td>γ.a.λ, 80K</td>
<td>γ.a.λ, Toys</td>
</tr>
</tbody>
</table>

(b) A tuple

**Figure 2.9.1 Skeleton and tuple**

Suppose each object has Name, which is the key attribute, Salary, and DName, which stands for
the department name of this object. How would we represent all the properties of an object under the skeleton model? Recall that it is the concurrence applied to identity which helps form the skeleton, and that such concurrence applies only to identity without extending to other properties of objects. With the assumption of homogeneity, same skeleton structure will apply to remaining attributes. As an example, Figure 2.9.1 (a) is the skeleton representation of name, salary and department name values, and Figure 2.9.1 (b) is the tabular representation of corresponding skeletons. This gives us a tuple. Note that identity as well as other properties can vary from subject to subject.

<table>
<thead>
<tr>
<th></th>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>γ, Jack</td>
<td>γ, 100K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td></td>
<td>γ, Jim</td>
<td>γ, 60K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td></td>
<td>γ, Jose</td>
<td>γ, 40K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td></td>
<td>γ, Jack</td>
<td>γ, 60K</td>
<td>γ, Clothes</td>
</tr>
<tr>
<td></td>
<td>γ, Jack</td>
<td>γ, 60K</td>
<td>γ, Clothes</td>
</tr>
<tr>
<td></td>
<td>γ, Jackson</td>
<td>γ, 80K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td>t₂</td>
<td>γ, Hari</td>
<td>γ, 50K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td></td>
<td>γ, Jun</td>
<td>γ, 70K</td>
<td>γ, PCs</td>
</tr>
<tr>
<td>t₃</td>
<td>α, Shyam</td>
<td>α, 50K</td>
<td>α, Toys</td>
</tr>
<tr>
<td></td>
<td>α, Jun</td>
<td>α, 70K</td>
<td>α, PCs</td>
</tr>
<tr>
<td>t₄</td>
<td>α, Jack</td>
<td>α, 40K</td>
<td>α, Shoes</td>
</tr>
<tr>
<td>t₅</td>
<td>γ, Lan</td>
<td>γ, 65K</td>
<td>γ, Shoes</td>
</tr>
<tr>
<td></td>
<td>γ, Ying</td>
<td>γ, 65K</td>
<td>γ, Clothes</td>
</tr>
<tr>
<td>t₆</td>
<td>β, Olga</td>
<td>β, 50K</td>
<td>β, PCs</td>
</tr>
<tr>
<td>t₇</td>
<td>β, Jackson</td>
<td>β, 60K</td>
<td>β, Clothes</td>
</tr>
<tr>
<td></td>
<td>β, Jackson</td>
<td>β, 80K</td>
<td>β, Toys</td>
</tr>
</tbody>
</table>

**Figure 2.9.2 Emp relation**

Every attribute in a tuple is a C-assignment. As a formal definition, a C-tuple, or simply a tuple is a concatenation of C-assignment whose security domains are the same, i.e. all the attribute values in the tuple have the same domain, and all the attribute values which have the same domain or path expression represent the properties of the same object. So the tuples in our model are
homogeneous [3].

Figure 2.9.2 is a tabular representation of all the tuples in Employee relation, where \( t_i \) is the \( i^{th} \) tuple. Objects and attribute values come from Figure 2.2.1 and concurrences come from Figure 2.3.2 (a). We can see that in \( t_i \), Jim at \( \alpha \) level, Jose at \( \beta \) level and Jack at \( \lambda \) level are known as Jack to \( \gamma \). Jack at \( \lambda \) level is known as Jose to \( \beta \). Jackson at \( \lambda \) level is known as Jim to \( \alpha \).

There are four properties about a tuple:

1. Tuple and skeleton are in 1-1 correspondence.
2. In each tuple, there won’t be two similar path expressions. This will imply a conflict of a user’s belief.
3. In every tuple, the path expressions are closed under prefixes.
4. In every tuple, every object is rooted on the root of the corresponding skeleton. The path expression for a root contains only one user level.

### 2.10 Dealing with masked path expressions between skeletons

You may notice that in Example 5, there are no two similar but not same path expressions in \( P_1 \) only or \( P_2 \) only (recall that \( P_1 = \{ \gamma, \gamma.a, \gamma.\lambda, \gamma.a.\lambda \} \), \( P_2 = \{ \gamma.\|, \gamma.a.\|, \gamma.\lambda, \gamma.a.\lambda \} \)). This is true for any tuple. In each tuple, a path expression is either masked or unmasked. There won’t be two similar path expressions existing in one tuple. The operations of union, intersection, difference, and complementation that we defined in definition 2 are operations in tuple level. In another words, these operations are aiming at dealing with generated tuples from the same skeleton. If a set of path expressions has similar but not same path expressions, this set contains information of more than one skeleton. In such case, this set can be broken down into several subsets, each mapping to
one skeleton. By using definition 2, we can deal with each skeleton easily. Different skeletons in a relation will be regarded as disjoint sets. Union, intersection, and difference between two different skeletons will be same as literal union, intersection, difference and complementation.

### 2.11 Compatible tuples

In our skeleton-based MLS model, we always store only the whole information in database from the highest user’s view. The information visible to any lower user is a subset of this information. There is no need to duplicate this information. Within each view (generated or original), the result of a query is nothing but this view with some objects masked. Whenever a query happens, according to who is asking the question, the first thing is to compute the user’s belief world. The second thing is to generate the result from this space.

We say two tuples are *compatible* if they have compatible key values. When two tuples are compatible, the union of them will result in one tuple. This is obvious when we think in terms of skeletons. These tuples are either from the original database, or generated from the database, they are the whole skeleton itself or part of the same skeleton. After the operation of union, they should maintain in one skeleton, in another word, one tuple.

### 2.12 Belief consistency among tuples

Recall in Figure 2.3.2 (a), the object Jackson at $\lambda$ level stays in two skeletons: one is a cascading view of Jack at $\gamma$ level; another is a cascading view of Jackson at $\beta$ level. Since we represent each skeleton as a tuple, Jackson at $\lambda$ level will reside in two tuples. If the salary of Jackson is 70K in one tuple and 80K in another, the problem of belief inconsistency will arise.
Suppose \( \tau_1 \) is a tuple belonging to a user \( u_1 \), \( \tau_2 \) is a tuple belonging to a lower user \( u_2 \), \( p_1 \) is a path expression in \( \tau_1 \) and \( p_2 \) is a path expression in \( \tau_2 \), such that for any \( p_1 \) and \( p_2 \), if one of \( p_1 \) and \( p_2 \) is the suffix of the other, and the key attribute in \( \tau_1 \) on path expression \( p_1 \) is same as key attribute in \( \tau_2 \) on path expression \( p_2 \), then the two tuples are said to be belief consistent if they agree on all attributes having the path expressions \( p_1 \) in \( \tau_1 \) and \( p_2 \) in \( \tau_2 \).

Belief inconsistency might happen when one object is saved in more than one tuples in database. In our model, it only happens on objects sitting where different skeletons meet. Whenever there is an update operation, we need to take care of belief consistency.

Note that the concept of belief consistency is different from the concept of compatible tuples. Belief inconsistency arises when one object has to be saved in different tuples in database, whereas, for all compatible tuples, they are generated in computation time from the same tuple which is saved in the database. There will not be any inconsistency in compatible tuples. If two tuples are incompatible, they must have been generated from two different tuples, i.e., they come from two different skeletons.
3. Extract algorithm: generating user’s view

Extract algorithm is used to compute a user’s belief world. In database, we always maintain the data according to the highest user’s view so that we could always keep a complete knowledge of data. Whenever a query happens, the first thing the system will do is to find out who is asking the question. According to the user’s security level, the system extracts all the objects and concurrences the user could see and form the tuples based on the tuple properties.

There are three steps of extract algorithm:

1. Throwing out the objects not belonging to the users above the current user level. This will leave us all the objects that the current user can see. Recall in a path expression, the last user level tells us where the object resides. Figure 3.1 (a) gives us the result of generating α’s view from γ’s view in Figure 2.9.1 after the first step in both tabular representation and graph representation. Notice that we throw away the tuples t₆ in Figure 2.9.1 and some objects inside other tuples, such as t₁.

2. Cleaning prefixes of domain so that the current user has no information about levels not belonging to him. The domain of objects within a tuple should be closed under prefixes and every point in the domain is rooted in the root of skeleton. As we are cleaning the prefixes of domain, some path expressions are reduced into just one user level. This results in more than 2 roots in one tuple (recall that only root has a path expression with just one user level). If we meet with such kind of path expression, we will move the object(s) from this path expression, as well as all the object(s) with path expressions having this path expression as prefix, into a new tuple. This is easily understood when we
think about a skeleton. After the domain restriction to a lower user, some objects at different levels which used to be in one skeleton might be broken into different skeletons (e.g. when we have to chop the root out). As a result, these objects will be broken into different tuples. From the point where the skeleton splits, all the descendents of this object will go with this object and form a new skeleton. In Figure 3.1 (b), after cleaning the prefixes, we have 7 tuples. The last tuple t’ is the one generated from previous t_1.

(3) Merging redundant objects. Objects which used to be meeting points of different skeletons and resided in different tuples now might still appear in several tuples, some of them becoming roots. We need to merge these objects together since they actually represent the same object. The rule of merging is always to merge an object being a root as well as all its descendents into the same object being a descendant of some other root.

In Figure 3.1 (c), we merge (λ, Jun) into t_2 and (λ, Jackson) into t_1.

Recall Figure 2.3.2 (b) is a scenario for concurrences from user α’s view. Compared with Figure 2.3.2 (b) and Figure 3.1 (c), they are consistent.
<table>
<thead>
<tr>
<th>Time (t)</th>
<th>NAME (γ, α)</th>
<th>SALARY (γ, α)</th>
<th>DNAME (γ, α)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>Jim</td>
<td>γ, α, 60K</td>
<td>Toys</td>
</tr>
<tr>
<td></td>
<td>Jack</td>
<td>γ, λ, 60K</td>
<td>Clothes</td>
</tr>
<tr>
<td></td>
<td>Jack</td>
<td>γ, β, 60K</td>
<td>Clothes</td>
</tr>
<tr>
<td></td>
<td>Jackson</td>
<td>γ, α, λ, 80K</td>
<td>Toys</td>
</tr>
<tr>
<td>t₂</td>
<td>Jun</td>
<td>γ, λ, 70K</td>
<td>PCs</td>
</tr>
<tr>
<td>t₃</td>
<td>Shyam</td>
<td>α, 50K</td>
<td>Toys</td>
</tr>
<tr>
<td></td>
<td>Jun</td>
<td>α, λ, 70K</td>
<td>PCs</td>
</tr>
<tr>
<td>t₄</td>
<td>Jack</td>
<td>α, 40K</td>
<td>Shoes</td>
</tr>
<tr>
<td>t₅</td>
<td>Ying</td>
<td>γ, α, 65K</td>
<td>Clothes</td>
</tr>
<tr>
<td>t₇</td>
<td>Jackson</td>
<td>β, λ, 80K</td>
<td>Toys</td>
</tr>
</tbody>
</table>

Figure 3.1. Three steps of generating α’s view from γ’s view
Figure 3.1 (Cont.) Three steps of generating $\alpha$’s view from $\gamma$’s view
4. Algebra

In parametric approach, algebra contains three parts: domain expressions, which evaluate to parametric elements, relational expressions, which evaluate multilevel security relations and boolean expressions, which evaluate a tuple to TRUE or FALSE.

Remember that the set of parametric elements is $C = \{\gamma, \alpha, \beta, \lambda, \gamma . \alpha, \gamma . \beta, \gamma . \lambda, \alpha . \lambda, \beta . \lambda, \gamma . \alpha . \lambda, \gamma . \beta . \lambda\}$. For any fixed user, all algebraic computations will be performed according to this user's view. The set $C$ contains all the possible valid parametric elements, which is also from the highest user's view. By putting restriction of domain, we could easily get concurrences for different users. For example, $C = \{\gamma, \alpha, \beta, \lambda, \gamma . \alpha, \gamma . \beta, \gamma . \lambda, \alpha . \lambda, \beta . \lambda, \gamma . \alpha . \lambda, \gamma . \beta . \lambda\}$ is also the parametric space that user $\gamma$ can see; $\{\alpha, \lambda, \alpha . \lambda\}$ is what $\alpha$ can see; $\{\beta, \lambda, \beta . \lambda\}$ is what $\beta$ can see; and $\{\lambda\}$ is what $\lambda$ can see.
<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl</td>
<td>y, Jack</td>
<td>y, 100K</td>
</tr>
<tr>
<td>t.a</td>
<td>y.a, Jim</td>
<td>y.a, 60K</td>
</tr>
<tr>
<td>t.b</td>
<td>γ, Jose</td>
<td>γ.b, 40K</td>
</tr>
<tr>
<td>t.λ</td>
<td>γ, Jack</td>
<td>γ.λ, 60K</td>
</tr>
<tr>
<td>t.βλ</td>
<td>γ, José</td>
<td>γ.βλ, 60K</td>
</tr>
<tr>
<td>t.αλ</td>
<td>γ, Jackson</td>
<td>γ.αλ, 80K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>t.γ</td>
<td>γ, Hari</td>
<td>γ, 50K</td>
</tr>
<tr>
<td>t.λ</td>
<td>γ, Jun</td>
<td>γ.λ, 70K</td>
</tr>
<tr>
<td>t.α</td>
<td>α, Shyam</td>
<td>α, 50K</td>
</tr>
<tr>
<td>t.λα</td>
<td>α, Jun</td>
<td>α.λ, 70K</td>
</tr>
<tr>
<td>t.α</td>
<td>α, Jack</td>
<td>α, 40K</td>
</tr>
<tr>
<td>t.γ</td>
<td>γ, Lan</td>
<td>γ, 65K</td>
</tr>
<tr>
<td>t.α</td>
<td>α, Ying</td>
<td>γ.α, 65K</td>
</tr>
<tr>
<td>t.β</td>
<td>β, Olga</td>
<td>β, 50K</td>
</tr>
<tr>
<td>t.βλ</td>
<td>β, Jackson</td>
<td>β, 60K</td>
</tr>
<tr>
<td>t.αλ</td>
<td>β, Jackson</td>
<td>β.λ, 80K</td>
</tr>
</tbody>
</table>

(a) Emp relation

<table>
<thead>
<tr>
<th>DNAME</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>t.1</td>
<td>γ, Toys</td>
</tr>
<tr>
<td>t.γ</td>
<td>γ.a, Toys</td>
</tr>
<tr>
<td>t.β</td>
<td>γ.b, Toys</td>
</tr>
<tr>
<td>t.λ</td>
<td>γ.λ, Toys</td>
</tr>
<tr>
<td>t.βλ</td>
<td>γ.βλ, Toys</td>
</tr>
<tr>
<td>t.αλ</td>
<td>γ.αλ, Toys</td>
</tr>
<tr>
<td>t.2</td>
<td>γ, Shoes</td>
</tr>
<tr>
<td>t.α</td>
<td>γ.a, Shoes</td>
</tr>
<tr>
<td>t.3</td>
<td>α, Clothes</td>
</tr>
<tr>
<td>t.λα</td>
<td>α.λ, Clothes</td>
</tr>
<tr>
<td>t.4</td>
<td>β, Clothes</td>
</tr>
<tr>
<td>t.βλ</td>
<td>β.λ, Clothes</td>
</tr>
<tr>
<td>t.3</td>
<td>β, PCs</td>
</tr>
<tr>
<td>t.βλ</td>
<td>β, PCs</td>
</tr>
</tbody>
</table>

(b) Dept relation

Figure 4.1. The tabular representation and skeleton representation of personnelDB
4.1 Domain expressions

Domain expressions are the syntactic counterpart of security elements. They are formed by using parametric elements, \([A]\), \([A0B]\), \([A0b]\), \([E]\), \(\cap\), \(\cup\), and \(-\), where \(A\) and \(B\) are attributes, \(b\) is a constant, and \(E\) is a relational expression. If \(\mu\) is a domain expression and \(\tau\) is a tuple then \(\mu(\tau)\) is the substitution of \(\tau\) in \(\mu\) and evaluates to a parametric element. We use symbol “\(\downarrow\)" to denote restriction in domain.

Example 7. Consider the domain expression \([\text{SALARY} = 60K]\). For a given tuple, the expression retrieves the path expression(s) where salary is 60K. For \(t_i\) in Emp relation in Figure 4.1,
[[SALARY=60K]](t_i) evaluates to \{γ\|, γ.α, γ.β\|, γ.λ, γ.β.λ, γ.α.λ\|\}. Consider the domain expression \[[SALARY=60K]\]∩¬[[DNAME=Clothes]]\}. For a given tuple, the expression retrieves the path expression(s) where salary is 60K and the department is not Clothes. For \(t_i\) in Emp relation in Figure 4.1, it evaluates to \{γ\|, γ.α, γ.β\|, γ.λ\|, γ.β.λ\|, γ.α.λ\|\}.

### 4.2 Boolean expressions

Boolean expressions evaluate a tuple to boolean values (TRUE or FALSE). They are formed by using \(\mu \leq \nu\), \(\wedge\), \(\vee\), and \(\neg\), where \(\mu\) and \(\nu\) are domain expressions.

### 4.3 Relational expressions

Relational expressions evaluate multilevel security relations.

#### 4.3.1. Union and difference

If \(r\) and \(s\) are relations over the schema \(R\) and with the key \(K\), then \(r ∪ s\) and \(r − s\) also have the same schema and key [3].

In case of union, compatible tuples across the two relations (recall that compatible tuples have same root) will be collapsed into one tuple. Compatible objects in these two consistent tuples will be collapsed together according to definition 3. Other tuples of \(r\) and \(s\) remain unaltered in \(r ∪ s\). In case of difference, the computation for compatible tuples will be done according to definition 2 and definition 3, and other tuples of \(r\) remain unchanged in \(r − s\).

#### 4.3.2. Projection

In the parametric model, a user thinks in terms of relations with keys[4]. In traditional relational databases, if \(r\) is a relation over the schema \(R\), and \(X\) is a set of attributes, the operator \(\pi_X(R)\)
defined by \( \pi_{X}(R) = \{\tau[X]: \tau \in R\} \) is a unary operation, and it might lack of the key. For example, \( \pi_{\text{SALARY}}(\text{Emp}) \) from Figure 4.1 in traditional databases will give us a set of tuples with only SALARY field which lacks a key. Thus, the field SALARY itself will become the key. As a result, the system will do the restructuring and eliminate the duplicate tuples. In multilevel security database, subjects have beliefs about the identities. And we assume that subjects only have beliefs about the identities of the objects; such beliefs don’t extend to other properties. We have no idea how to deal with restructuring without a key. To solve the problem, whenever we do the projection, we will always bring the key attribute in the projection result to keep the subjects’ beliefs.

If \( r \) is a relation over the schema \( R \) and with the key \( K \), and \( X \) is a set of attributes, \( \pi_{X}(R) = \{\tau[K,X]: \tau \in R\} \). Figure 4.2 shows the result of \( \pi_{\text{SALARY}}(\text{Emp}) \).

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>( \gamma, ) Jack ( \gamma, 100K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \alpha, ) Jim ( \gamma, \alpha, 60K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \beta, ) Jose ( \gamma, \beta, 40K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \lambda, ) Jack ( \gamma, \lambda, 60K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \beta, \lambda, ) Jack ( \gamma, \beta, \lambda, 60K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \alpha, \lambda, ) Jackson ( \gamma, \alpha, \lambda, 80K )</td>
</tr>
<tr>
<td>t₂</td>
<td>( \gamma, ) Hari ( \gamma, 50K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \lambda, ) Jun ( \gamma, \lambda, 70K )</td>
</tr>
<tr>
<td>t₃</td>
<td>( \alpha, ) Shyam ( \alpha, 50K )</td>
</tr>
<tr>
<td></td>
<td>( \alpha, \lambda, ) Jun ( \alpha, \lambda, 70K )</td>
</tr>
<tr>
<td>t₄</td>
<td>( \alpha, ) Jack ( \alpha, 40K )</td>
</tr>
<tr>
<td>t₅</td>
<td>( \gamma, ) Lan ( \gamma, 65K )</td>
</tr>
<tr>
<td></td>
<td>( \gamma, \alpha, ) Ying ( \gamma, \alpha, 65K )</td>
</tr>
<tr>
<td>t₆</td>
<td>( \beta, ) Olga ( \beta, 50K )</td>
</tr>
<tr>
<td>t₇</td>
<td>( \beta, ) Jackson ( \beta, 60K )</td>
</tr>
<tr>
<td></td>
<td>( \beta, \lambda, ) Jackson ( \beta, \lambda, 80K )</td>
</tr>
</tbody>
</table>

**Figure 4.2** \( \pi_{\text{SALARY}}(\text{Emp}) \)
4.3.3. Selection

A selection is of the form $\sigma(r, f, \mu)$, where $f$ is a boolean expression and $\mu$ is a domain expression. The selection $\sigma(r, f, \mu)$ evaluates to $\{x \text{ restricted to } \mu : f(x)\}$. The key of $\sigma(r, f, \mu)$ is the same as the key of $r$. If $f$ evaluates to TRUE for a tuple, $\sigma$ allows a user to select only a relevant part of it, which is specified by $\mu$. Especially in our model, if the whole tuple is not relevant, this tuple will be discarded; otherwise, the path expressions for relevant part of this tuple are unmasked, and for the irrelevant part of it are all masked. In another word, whenever we select a part of a skeleton, we will retrieve all objects in this skeleton except that relevant objects are unmasked and irrelevant objects are masked. By keeping the masked objects, we keep the path expressions closed under prefixes.

Example 8. $\gamma$ asks the information about employees who are thought to work in Toys by $\alpha$.

$\sigma(\text{Emp}, [\text{NAME\downarrow.*}] \cap [\text{DNAME=Toys}])$

![Skeleton representation](image.png)

Figure 4.3. The query result of Example 8
<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ</td>
<td></td>
<td>J</td>
</tr>
<tr>
<td>γ,a,J</td>
<td>Jim</td>
<td>γ,a, 60K</td>
</tr>
<tr>
<td>γ,b,J</td>
<td>Jose</td>
<td>γ,b, 40K</td>
</tr>
<tr>
<td>γ,λ,J</td>
<td>Jack</td>
<td>γ,λ, 60K</td>
</tr>
<tr>
<td>γ,b,λ,J</td>
<td>Jack</td>
<td>γ,b,λ, 60K</td>
</tr>
<tr>
<td>γ,a,λ,</td>
<td>Jackson</td>
<td>γ,a,λ, 80K</td>
</tr>
<tr>
<td>α</td>
<td>Shyam</td>
<td>α, 50K</td>
</tr>
<tr>
<td>α,λ,J</td>
<td>Jun</td>
<td>α,λ, 70K</td>
</tr>
</tbody>
</table>

(b) Tabular representation

Figure 4.3 (Cont.) The query result of Example 8

Figure 4.3 shows the result of query in Example 8. Objects which are circled are masked objects in skeleton representation. When we compute this query, the first thing we need to do is get the user’s γ view since it’s γ who is asking the question. This validates the maximum data set we could work on. Since there is no boolean expression, the second thing is to evaluate the domain expression \([\text{NAME}^* \cdot \alpha ^*] \cap [\text{DNAME}=\text{Toys}]\). After evaluating the restriction \([\text{NAME}^* \cdot \alpha ^*]\), we actually get three tuples, besides the two tuples in Figure 4.3 (b), we also had the following tuple.

| α, Jack | α, 40K | α, Shoes |

But then we discarded this tuple since Jack doesn’t work in Toys department. Finally we make all the irrelevant objects masked.

Example 9. α asks the information about employees who are thought to work in Toys by α.

σ(Emp, \([\text{NAME}^* \cdot \alpha ^*] \cap [\text{DNAME}=\text{Toys}])\)
Figure 4.4. The query result for Example 9

The difference of Example 8 and Example 9 comes from the context of user levels. Different users have different views which will be generated in the first step in computation time.
5. Update mechanism

In Skeleton-based MLS model, a user can update objects only at its own level. To preserve belief consistency, the update for each object has to validate the same object existing in other tuples. This action will be taken inside the system, user won’t be aware of it.

In the following we use K as an abbreviation to denote the key of a relation r which might actually be multiple keys K_1, K_2, ..., K_n and K=k to denote an assignment of ordinary values to keys. For any attribute M, M=m denotes assignment to an arbitrary set M of attributes of r.

5.1 Connect clause and asserting a belief

Connect clause is used for a user to allow lower object(s) to be connected. The format of connect clause is “connect [{u_1: (K=k_1)}] to {u_2: (K=k_2)}”, where k_1 is the key values of the current user, “u_1” is the higher user, “u_2” is the lower user and “k_2” is the key values of the lower object. Notice that only the key values are specified. Values of other attributes are determined automatically.

The part “[{u_1: (K=k_1)}]” in connect clause is optional. When connect clause is included in other update operations, such as insert or update, this part can be omitted.

If execution of a connect clause succeeds, a concurrence will be generated. When executing the connect clause, the system will first check whether the concurrence that will be generated from connect clause is valid or not. If not, the system will not proceed and return an error. Otherwise, the path expressions of related objects will be changed accordingly.

Example 10. (For γ) Suppose γ wants to connect Lan in its own level to Jackson in β level in Emp
relation, the corresponding connect clause and the result are in Figure 5.1:

connect (γ, Lan) to (β, Jackson)

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ, Lan</td>
<td>γ, 65K</td>
<td>γ, Shoes</td>
</tr>
<tr>
<td>γ, a, Ying</td>
<td>γ, a, 65K</td>
<td>γ, a, Clothes</td>
</tr>
<tr>
<td>β, Jackson</td>
<td>β, 60K</td>
<td>β, Clothes</td>
</tr>
<tr>
<td>β, λ, Jackson</td>
<td>β, λ, 80K</td>
<td>β, λ, Toys</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ, Lan</td>
<td>γ, 65K</td>
<td>γ, Shoes</td>
</tr>
<tr>
<td>γ, a, Ying</td>
<td>γ, a, 65K</td>
<td>γ, a, Clothes</td>
</tr>
<tr>
<td>γ β, Jackson</td>
<td>γ β, 60K</td>
<td>γ β, Clothes</td>
</tr>
<tr>
<td>γ β, λ, Jackson</td>
<td>γ β, λ, 80K</td>
<td>γ β, λ, Toys</td>
</tr>
</tbody>
</table>

Figure 5.1. The query result for Example 10

After execution of connect clause, tuples t5 and t7 are collapsed into one tuple. The path expressions previously in t7 now are starting from γ, showing the belief path that starts from γ.

5.2 Disconnect clause and removing a belief

Disconnect clause is used for a user to allow lower object(s) to be disconnected. The format of disconnect clause is “disconnect [{u1: (K=k1)}] from {u2: (K=k2)}” which means “disconnect the object whose key values are k1 at the user level u1 and the object whose key values are k2 at u2 level”. In this clause, the user who wants to perform this action is always u1, which is the higher level user. Same as connect clause, the values of attributes other than the key values will be determined automatically.

Disconnect clause happens in one tuple. After executing the disconnect clause, the tuple will be broken down into several separate tuples. The path expressions within each tuple will be altered
accordingly.

Same as connect clause, disconnect clause can also be nested into other operations. In that case, the part \( \{[u_1: (K=k_1)]\} \) can be omitted.

**Example 11.** User \( \gamma \) wants to disconnect Jack in its own level from Jim in a level in Emp relation, the corresponding disconnect clause and the result are in Figure 5.2:

\[
\text{disconnect (}\gamma, \text{ Jack) from (}\alpha, \text{ Jim)}
\]

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma, ) Jack</td>
<td>( \gamma, 100K )</td>
<td>( \gamma, ) Toys</td>
</tr>
<tr>
<td>( \gamma, \alpha, ) Jim</td>
<td>( \gamma, \alpha, 60K )</td>
<td>( \gamma, \alpha, ) Toys</td>
</tr>
<tr>
<td>( \gamma, \beta, ) Jose</td>
<td>( \gamma, \beta, 40K )</td>
<td>( \gamma, \beta, ) Toys</td>
</tr>
<tr>
<td>( \gamma, \lambda, ) Jack</td>
<td>( \gamma, \lambda, 60K )</td>
<td>( \gamma, \lambda, ) Clothes</td>
</tr>
<tr>
<td>( \gamma, \beta, \lambda, ) Jack</td>
<td>( \gamma, \beta, \lambda, 60K )</td>
<td>( \gamma, \beta, \lambda, ) Clothes</td>
</tr>
<tr>
<td>( \gamma, \alpha, \lambda, ) Jackson</td>
<td>( \gamma, \alpha, \lambda, 80K )</td>
<td>( \gamma, \alpha, \lambda, ) Toys</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma, ) Jack</td>
<td>( \gamma, 100K )</td>
<td>( \gamma, ) Toys</td>
</tr>
<tr>
<td>( \gamma, \beta, ) Jose</td>
<td>( \gamma, \beta, 40K )</td>
<td>( \gamma, \beta, ) Toys</td>
</tr>
<tr>
<td>( \gamma, \lambda, ) Jack</td>
<td>( \gamma, \lambda, 60K )</td>
<td>( \gamma, \lambda, ) Clothes</td>
</tr>
<tr>
<td>( \gamma, \beta, \lambda, ) Jack</td>
<td>( \gamma, \beta, \lambda, 60K )</td>
<td>( \gamma, \beta, \lambda, ) Clothes</td>
</tr>
</tbody>
</table>

Related tuple before executing disconnect clause

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha, ) Jim</td>
<td>( \alpha, 60K )</td>
<td>( \alpha, ) Toys</td>
</tr>
<tr>
<td>( \alpha, \lambda, ) Jackson</td>
<td>( \alpha, \lambda, 80K )</td>
<td>( \alpha, \lambda, ) Toys</td>
</tr>
</tbody>
</table>

Generated tuples after executing disconnect clause

**Figure 5.2. The query result for Example 11**

### 5.3 Insert

When a user \( u \) inserts an object, the user may also want to connect it to identities at lower levels. The first thing system will do is to check if the new object exists at level \( u \). If so, the user will get an error from the system; otherwise, the insert transaction will proceed. When it comes to connect, the same procedure as mentioned in 5.1 will be proceeded. The format of insert is
Insert \((X=x)\) in \(r\)
\[\text{connect to } \{u: (K=k)\}\]

**Example 12.** User \(\gamma\) inserts the object “Mike” in the Emp relation and associates the identity to “Jackson” at \(\beta\) level. The corresponding insert clause and the result are in Figure 5.3:

\[
\text{insert } (\text{NAME} = \text{Mike}; \text{SALARY} = 100K; \text{DNAME} = \text{Toys}) \text{ in Emp}
\]
\[
\text{connect to } (\beta, \text{Jackson})
\]

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta), Jackson</td>
<td>(\beta, 60K)</td>
<td>(\beta, \text{Clothes})</td>
</tr>
<tr>
<td>(\beta, \lambda), Jackson</td>
<td>(\beta, \lambda, 80K)</td>
<td>(\beta, \lambda, \text{Toys})</td>
</tr>
</tbody>
</table>

Before executing the insertion

| \(\gamma, \text{Mike}\) | \(\gamma, 100K\) | \(\gamma, \text{Toys}\) |
| \(\gamma, \beta, \text{Jackson}\) | \(\gamma, \beta, 60K\) | \(\gamma, \beta, \text{Clothes}\) |
| \(\gamma, \beta, \lambda, \text{Jackson}\) | \(\gamma, \beta, \lambda, 80K\) | \(\gamma, \beta, \lambda, \text{Toys}\) |

After executing the insertion

*Figure 5.3. The query result for Example 12*

**5.4 Delete**

A user can only delete an object in its own object space. When a user \(u\) deletes an object, if this object exists in multiple tuples, the identity of the object will be removed from each tuple in the relation. The system will also remove the concurrence related to this object. This might break one tuple into several tuples. If the generated tuple is a subset of some other tuple, they will be merged together. If a tuple becomes empty, it will be removed from the relation. The delete operation is of the form

\[
\text{delete } (K=k) \text{ from } r
\]

**Example 13.** User \(\alpha\) deletes the object “Jim” from the Emp relation. The corresponding delete
clause and the result are in Figure 5.4:

delete (NAME=Jim) from Emp

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ, Jack</td>
<td>γ, 100K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td>γ.α, Jim</td>
<td>γ.α, 60K</td>
<td>γ.α, Toys</td>
</tr>
<tr>
<td>γ.β, Jose</td>
<td>γ.β, 40K</td>
<td>γ.β, Toys</td>
</tr>
<tr>
<td>γ.λ, Jack</td>
<td>γ.λ, 60K</td>
<td>γ.λ, Clothes</td>
</tr>
<tr>
<td>γ.β.λ, Jack</td>
<td>γ.β.λ, 60K</td>
<td>γ.β.λ, Clothes</td>
</tr>
<tr>
<td>γ.α.λ, Jackson</td>
<td>γ.α.λ, 80K</td>
<td>γ.α.λ, Toys</td>
</tr>
</tbody>
</table>

Before executing the deletion

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>γ, Jack</td>
<td>γ, 100K</td>
<td>γ, Toys</td>
</tr>
<tr>
<td>γ.β, Jose</td>
<td>γ.β, 40K</td>
<td>γ.β, Toys</td>
</tr>
<tr>
<td>γ.λ, Jack</td>
<td>γ.λ, 60K</td>
<td>γ.λ, Clothes</td>
</tr>
<tr>
<td>γ.β.λ, Jack</td>
<td>γ.β.λ, 60K</td>
<td>γ.β.λ, Clothes</td>
</tr>
</tbody>
</table>

After executing the deletion

**Figure 5.4. The query result for Example 13**

Note that after deleting Jim at α level, Jackson at λ level didn’t form a new tuple. This is because Jackson at λ level is a subset of the tuple t₁ in original Emp relation. It’s merged into t₇.

### 5.5 Update

A user can only update the object in its own object space. There will be two different situations of update operations: update key values and update attribute values which are not key values. We only talk about update of attribute values which are not key values in this paper. Update of key values will be our future work. The update is of the form

```
update (K=k) in r to (X=x)
[connect to {u: (K'=k')}}]
```

When update of an object happens, each tuple involving the identity will be updated. Connect
clause may be nested. If the concurrence that will be generated after the connect clause violates the existing concurrence, the system will only do the update of the attribute values and report the error of connect operation. In this case, if user wants to change the belief, he has to disconnect the concurrence first and connect the object to some other object.

**Example 14.** User $\gamma$ wants to change the salary of Lan to 85K and then connect it to Jack. The corresponding update clause and the result are in Figure 5.5:

```
update (NAME=Lan) in Emp to (SALARY=85K)
connect to {\alpha:(NAME=Jack)}
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_4$</td>
<td>$\alpha$, Jack</td>
<td>$\alpha$, 40K</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$\gamma$, Lan</td>
<td>$\gamma$, 65K</td>
</tr>
<tr>
<td></td>
<td>$\gamma \alpha$, Ying</td>
<td>$\gamma \alpha$, 65K</td>
</tr>
</tbody>
</table>

**Before update**

<table>
<thead>
<tr>
<th>Name</th>
<th>SALARY</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_4$</td>
<td>$\alpha$, Jack</td>
<td>$\alpha$, 40K</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$\gamma$, Lan</td>
<td>$\gamma$, 85K</td>
</tr>
<tr>
<td></td>
<td>$\gamma \alpha$, Ying</td>
<td>$\gamma \alpha$, 65K</td>
</tr>
</tbody>
</table>

**After update**

*Figure 5.5. The query result for Example 14*

Note that even after the update, the concurrence from ($\gamma$, Lan) to ($\alpha$, Jack) is not formed. This is because system found that there is already a concurrence from ($\gamma$, Lan) to ($\alpha$, Ying), so building another concurrence from ($\gamma$, Lan) to ($\alpha$, Jack) is not legal. The system will report this information to the user $\gamma$. If $\gamma$ still wants to change the belief, he should first disconnect the existing concurrence and then build a new one.
6. Related works

Now we make some comparisons with [9]. In our model we assume that the object spaces at different levels are independent. [9] assumes that if a subject believes in existence of an object, all upper subjects must also believe in its existence as well. We assume that even when the upper subject does concur about existence, the identity of the object can be different at the upper level. On the other hand [9] assumes that the identity of an object will be same at all levels. This applies to subjects that are parallel as well as strictly higher. What about other attributes? We assume that these values are also independent at all levels. [9] gives a way to infer these values from values that exist at levels immediately below, sometimes leading to null values. [2] introduced updates where a upper user could either manually enter its own values for key, as well as non-key attributes. A notion of subscription was also developed, that allow different non-key attribute values to be propagated automatically from designated lower user. At present our work in multilevel security has not considered nulls.

Now we briefly discuss the type of nulls incorporated in [9]. Recall that the values of key attributes cannot have nulls. Therefore, we only consider a non-key attribute value of some object at some level. This value can be derived from values of the same object from levels directly below. If all levels directly below agree on a non-null value, this value can be propagated upward. When the levels differ, but have non-null values, either the non-null value belonging to one user or a null can be propagated. When there is a mix of non-null and null values there is option of giving priority to non-null values to propagate; otherwise a null is propagated. When all values at level directly below are null, a null is propagated. It is shown that depending upon the choices made,
the propagation is most informative and without leading to inconsistencies.

In addition to propagation of values, [9] also considers SQL-style primary keys and foreign-keys. Note that in SQL a foreign key in the referencing relation has to be a primary-key in the referenced relation. As expected, primary key attributes cannot have nulls. If a foreign key is non-null, it has to exist as a primary-key in the referenced relation. These constraints have been formally developed for the classical relations and applied at a fixed level at a time.

We talked about belief consistency, and paper [7] gave us a belief consistent MLS data model. Authors of [7] look at each object as a tuple as long as there is no different belief on this object from different users. If there is more than one different belief on a piece of data, the objects (even if they are viewed as same identity) will be kept in different tuples. The problem of belief consistency easily comes up in their model and a set of rules has to set up to preserve belief consistency. Coming along with this, query about different beliefs becomes complicated. In our model, since we maintain cascading beliefs in a tuple, we only take care of such situation in the points where different skeletons meet in any update operation. Our approach makes query of multiple beliefs efficient and largely reduces join.
7. Conclusion and future works

In this paper we provide a parametric approach to build an MSL model. Our Skeleton-based MLS model assembles cascading beliefs of an object. Each skeleton is represented as a tuple. By using path expressions as parametric elements, we preserve user’s cascading view of an object into one tuple. This relieves the user the burden of having to assemble cascading of beliefs through multiple joins. Based on this, we can design an efficient query language for users to ask questions about multiple beliefs. At the same time, join will be largely reduced. As an important property of path expressions within a tuple, closure under prefixes helps algebraic identities hold. In order to make sure path expressions are always closed under prefixes within a tuple, mask will be applied if necessary. The closure under prefixes of path expressions and masking/unmasking are managed by the system automatically.

We provided the algebra union, intersection, difference and projection as well as the update mechanism of the model. We are still working on the associative navigation A0B. After this, we are able to provide the natural join. When we finish all the algebra, we will provide an SQL-like query language and consider the operation of change of key attributes. This finishes the skeleton-based model. We will also try to consider about graph-based model, which will model all the connected points in one value. Recall that in skeleton-based model, several skeletons might be connected. In graph-based model, all these connected skeletons will be modeled in one value. By doing so, the problem of belief inconsistency will totally disappear. Further more, this model will handle even more complicated queries, such as query about atomic values in one skeleton from another skeleton through connected points.
References


