School of
Information Systems

SMU
SINGAPORE MANAGEMENT UNIVERSITY

A Robust Damage Assessment Model for Corrupted Database Systems

Ge Fu Research Assistant, SMU

Yingjiu Li Assistant Professor, SMU

Email: ***{gefu,yjli}@smu.edu.sg***

Outlines

- Background: Traditional damage assessment in database systems
- Motivation: Problems of existing damage assessment model
- Solution: Introducing new concepts and extending damage assessment model
- Conclusions and future work

Background

Traditional Damage Assessment in Database systems

Towards Database Security Ⅰ

- What does conventional database security mechanism concern?
  - confidentiality, integrity, availability, survivability
- What we have done?
  - Authentication & Authorization
  - Access Control (DAC, MAC, FGAC et al.)
  - Inference Control
  - Multilevel Secure Databases
  - Data Encryption

Towards Database Security Ⅱ

- Disadvantages of these Methods?
  - Addresses primarily how to protect the security of a database;
  - Preventative based methods can not prevent all attacks;
    - E.g. SQL injection & cross site script attacks
  - Damage Assessment for a post-intrusion database system becomes an important issue.

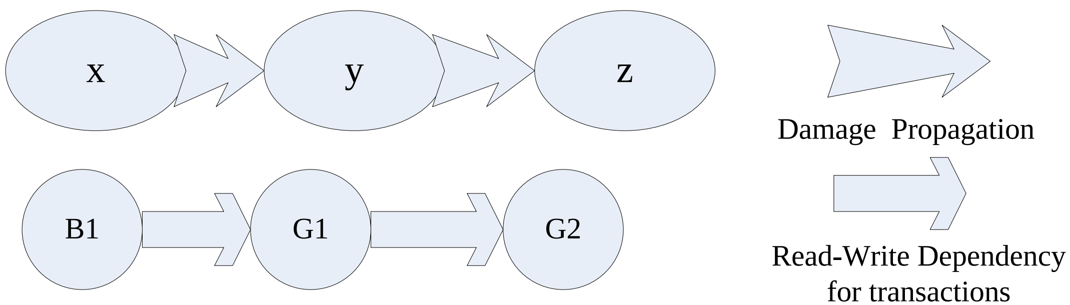
Traditional Damage Assessment

t1 — B1: x=x+2;
(Malicious transaction B1 modifies data item x)

t2 — G1: y=x+9;
(Benign transaction G1 reads x and modifies y)

t3 — G2: z=y+10;
(Benign transaction G2 reads y and modifies z)

time



when transaction B1 that updates x is identified malicious, the damage on x can spread to every data item updated by a transaction that is dependent on B1 directly or indirectly.

Traditional Damage Assessment

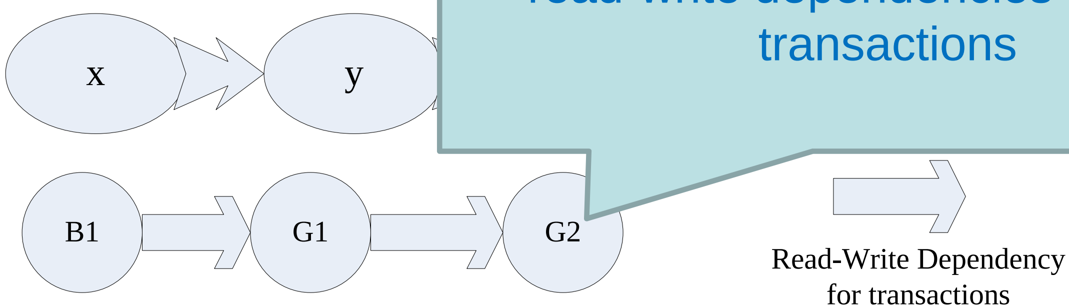
t1 — B1: x=x+2;
(Malicious transaction B1 modifies data item x)

t2 — G1: y=x+9;
(Benign transac

t3 — G2: z=y+10;
(Benign transac

time

Purpose of Damage Assessment is to locate the scope of damage propagation and affected transactions caused by read-write dependencies between transactions



when transaction B1 that updates x is identified malicious, the damage on x can spread to every data item updated by a transaction that is dependent on B1 directly or indirectly.

Motivation

Problems of existing damage assessment model

Inter-transaction Dependency Analysis

- Transaction Dependency relationships are caused by data sharing.
- There are four data sharing modes may cause a dependency relation from T_2 to T_1 ($T_1 <_H T_2$):
  - Read-Read mode
    - Transaction T_2 reads a data item x that is read by T_1
  - Read-Write mode
    - Transaction T_2 writes a data item x that is read by T_1
  - Write-Read mode
    - Transaction T_2 reads a data items x that is written by T_1
  - Write-Write mode
    - Transaction T_2 writes a data items x that is written by T_1

Inter-transaction Dependency Analysis

- Transaction Dependency relationships are caused by data sharing.
- There are four data sharing modes may cause a dependency relation from T_2 to T_1 ($T_1 <_H T_2$):
  - Read-Read mode
    - Transaction T_2 reads a d…
  - Read-Write mode
    - Transaction T_2 writes a data ite… hat is read by T_1
  - Write-Read mode
    - Transaction T_2 reads a data items x that is written by T_1
  - Write-Write mode
    - Transaction T_2 writes a data items x that is written by T_1

Read- Read mode and Read-Write mode cannot cause damage propagation and transaction dependency

Inter-transaction Dependency Analysis

- Transaction Dependency relationships are caused by data sharing.
- There are four data sharing modes may cause a dependency relation from T_2 to T_1 ($T_1 <_H T_2$):
  - Read-Read mode
    - Transaction T_2 reads a da
  - Read-Write mode
    - Transaction T_2 writes a data ite hat is read by T_1
  - Write-Read mode
    - Transaction T_2 reads a da
  - Write-Write mode
    - Transaction T_2 writes a d hat is written by T_1

Read- Read mode and Read-Write mode cannot cause damage propagation and transaction dependency

Write- Read mode: traditional read- write dependency
Write-Write mode: cause dependency between transaction?

Phantoms Dependency

Application Logic:
An increase of commodity prices: we need the price of commodity whose price is more than *$500* increased by *10%*.
Intrusion Activity:
Assume that before the price increase activity occurs, there is a malicious transaction B_1 modifies the product rice's price from *$400* to *$600*. The correspondent transaction history is described as follows:
Correlated Database Layer SQL statements:
G_0: UPDATE *product* SET $price = 600$ WHERE $product\_name =$ "*rice*";
B_1: UPDATE *product* SET $price = 400$ WHERE $product\_name =$ "*rice*";
G_1: UPDATE *product* SET $price = 1.1 * price$ WHERE $price > 500$;

There is no read-write dependency relation from G_1 to B_1

- G_1 does not read from B_1

Phantoms Dependency

Application Logic:
An increase of commodity prices: we need the price of commodity whose price is more than *\$500* increased by *10%*.
Intrusion Activity:
Assume that before the price increase activity occurs, there is a malicious transaction B_1 modifies the product rice's price from *\$400* to *\$600*. The correspondent transaction history is described as follows:
Correlated Database Layer SQL statements:
G_0: UPDATE *product* SET *price* = 600 WHERE *product_name* = "*rice*";
B_1: UPDATE *product* SET *price* = 400 WHERE *product_name* = "*rice*";
G_1: UPDATE *product* SET *price* = 1.1 * *price* WHERE *price* > 500;

There is no read-write dependency relation from G_1 to B_1

- G_1 does not read from B_1

Is G_1 affected by B_1?
Obviously yes!

- all products' prices increase by *10%,* except for the product *rice*!

Phantoms Dependency

Application Logic:
An increase of commodity prices: we need the price of commodity whose price is more than *\$500* increased by *10%*.
Intrusion Activity:
Assume that before the price increase activity occurs, there is a malicious transaction B_1 modifies the product rice's price from *\$400* to *\$600*. The correspondent transaction history is described as follows:
Correlated Database Layer SQL statements:
G_0: UPDATE *product* SET *price* = 600 WHERE *product_name* = "*rice*";
B_1: UPDATE *product* SET *price* = 400 WHERE *product_name* = "*rice*";
G_1: UPDATE *product* SET *price* = 1.1 ∗ *price* WHERE *price* > 500;

There is no read-write dependency relation from G_1 to B_1

- G_1 does not read from B_1

Is G_1 affected by B_1?
Obviously yes!

- all products' prices increase by *10%,* except for the product *rice*!

In the common sense, It seemed that G_1 *is affected by* B_1 *for* G_1 *read a phantom* data "*rice*" (means it has been deleted). We denote the dependency from G_1 *to* B_1 *as Phantoms Dependency.*

Pseudo-Identity Dependency

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P002	orange	$120
P003	apple	$100
P004	flour	$460

(a) Initial state of table *product*

Pseudo-Identity Dependency

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P002	orange	$120
P003	apple	$100
P004	flour	$460

(a) Initial state of table *product*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460

(b) Malicious transaction B deletes the record with *product_id="P002"*

Pseudo-Identity Dependency

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P002	orange	$120
P003	apple	$100
P004	flour	$460

(a) Initial state of table *product*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460

(b) Malicious transaction *B* deletes the record with *product_id="P002"*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460
P002	grape	$320

(c) innocent transaction *G* inserts a new record with *product_id="P002"*

School of
Information Systems

Pseudo-Identity Dependency

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P002	orange	$120
P003	apple	$100
P004	flour	$460

(a) Initial state of table *product*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460

(b) Malicious transaction *B* deletes the record with *product_id="P002"*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460
P002	grape	$320

(c) innocent transaction *G* inserts a new record with *product_id="P002"*

Is innocent transaction *G* affected by malicious transaction *B*?
Obviously, Yes!

- if the product *P002* was not deleted by transaction *B*, transaction *G* could not have been executed successfully!

Pseudo-Identity Dependency

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P002	orange	$120
P003	apple	$100
P004	flour	$460

(a) Initial state of table *product*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460

(b) Malicious transaction *B* deletes the record with *product_id="P002"*

product_id (PK)	name	price
P000	rice	$400
P001	banana	$230
P003	apple	$100
P004	flour	$460
P002	grape	$320

(c) innocent transaction *G* inserts a new record with *product_id="P002"*

Is innocent transaction G affected by malicious transaction B?
Obviously, Yes!

- if the product *P002* was not deleted by transaction *B,* transaction *G* could not have been executed successfully!

In our intuitive feeling, transaction *G* creates a new entity with a pseudo identity to substitute the historical object so as to satisfy the entity integrity constraint. We denote this kind of dependency from *G to B as Pseudo-Identity Dependency.*

Domain-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(a) Initial state of table *product*, where a domain integrity constraint $CHECK(p\_price < r\_price)$ is imposed on table *product*

Domain-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(a) Initial state of table *product*, where a domain integrity constraint $CHECK(p\_price < r\_price)$ is imposed on table *product*

product_id (PK)	p_price	r_price
P000	$350	$500
P001	$230	$246
P002	$460	$486

(b) Malicious transaction B updates the record with *product_id="P000"* and decreases *purchase_price* to *$350*

Domain-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(a) Initial state of table *product*, where a domain integrity constraint $CHECK(p\_price < r\_price)$ is imposed on table *product*

product_id (PK)	p_price	r_price
P000	$350	$500
P001	$230	$246
P002	$460	$486

(b) Malicious transaction B updates the record with *product_id="P000"* and decreases *purchase_price* to *$350*

product_id(PK)	p_price	r_price
P000	$350	$360
P001	$230	$246
P002	$460	$486

(c) Innocent transaction G *UPDATE* the record with *product_id="P000"* and decreases *retail_price* to *$360*

School of
Information Systems

SMU SINGAPORE MANAGEMENT UNIVERSITY

Domain-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(a) Initial state of table *product*, where a domain integrity constraint $CHECK(p\_price < r\_price)$ is imposed on table *product*

product_id (PK)	p_price	r_price
P000	$350	$500
P001	$230	$246
P002	$460	$486

(b) Malicious transaction *B* updates the record with *product_id="P000"* and decreases *purchase_price* to *$350*

product_id(PK)	p_price	r_price
P000	$350	$360
P001	$230	$246
P002	$460	$486

(c) Innocent transaction *G UPDATE* the record with *product_id="P000"* and decreases *retail_price* to *$360*

Is innocent transaction *G* affected by malicious transaction *B*?

Obviously, Yes!

- if *B did not exist, G would not have been executed* successfully due to the *CHECK* constraint imposed on table product (if *B* did not exist, *G* renewed *r_price* to *$360,* then the function *CHECK($400<$360)* would return a value *false).*

Domain-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(a) Initial state of table *product*, where a domain integrity constraint *CHECK*(*p_price* < *r_price*) is imposed on table *product*

product_id (PK)	p_price	r_price
P000	$350	$500
P001	$230	$246
P002	$460	$486

(b) Malicious transaction *B* updates the record with *product_id="P000"* and decreases *purchase_price* to *$350*

product_id(PK)	p_price	r_price
P000	$350	$360
P001	$230	$246
P002	$460	$486

(c) Innocent transaction *G UPDATE* the record with *product_id="P000"* and decreases *retail_price* to *$360*

Is innocent transaction *G* affected by malicious transaction *B*?

Obviously, Yes!

- if *B did not exist, G would not have been executed* successfully due to the *CHECK* constraint imposed on table product (if *B* did not exist, *G* renewed *r_price* to *$360,* then the function *CHECK($400<$360)* would return a value *false).*

This dependency relation from transaction *G to* B is caused by tuple-level domain-integrity constrains. We denote this kind of dependency as *Domain-Integrity Dependency*

Reference-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246

(a) Main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300

(b) Initial state of slave table *order*, where a foreign key constraint in which *order(p_id)* references to *product(product_id)* without any CASCADE policy

Reference-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246

(a) Main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300

(b) Initial state of slave table *order*, where a foreign key constraint in which *order(p_id)* references to *product(product_id)* without any CASCADE policy

Reference-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246

(a) Main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300

(b) Initial state of slave table *order*, where a foreign key constraint in which *order(p_id)* references to *product(product_id)* without any CASCADE policy

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(c) Malicious transaction *B* inserts a product with *product_id="P002"* into main table *product*

Reference-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246

(a) Main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300

(b) Initial state of slave table *order*, where a foreign key constraint in which *order(p_id)* references to *product(product_id)* without any CASCADE policy

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(c) Malicious transaction *B* inserts a product with *product_id="P002"* into main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300
O003	P002	260

(d) Innocent transaction *G* inserts a new order that references to product *P002*

Reference-Integrity Dependency

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246

(a) Main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300

(b) Initial state of slave table *order*, where a foreign key constraint in which *order(p_id)* references to *product(product_id)* without any CASCADE policy

product_id (PK)	p_price	r_price
P000	$400	$500
P001	$230	$246
P002	$460	$486

(c) Malicious transaction *B* inserts a product with *product_id="P002"* into main table *product*

order_id(PK)	p_id(FK)	quantity
O001	P000	500
O002	P001	300
O003	P002	260

(d) Innocent transaction *G* inserts a new order that references to product *P002*

Is innocent transaction *G* affected by malicious transaction *B*?

Obviously, Yes!

- transaction *G* inserts data that references to some other corrupted data. The damage propagates from the main table to the slave table. The innocent transaction is affected by malicious transaction. We denote this kind of dependency *Reference-Integrity Dependency.*

Solution

Introducing new concepts and extending damage assessment model

Phantoms Dependency

Definition 1 **Phantoms Dependency**. Consider a transaction history H:..., T_1, ..., T_2,... that satisfies:

1) there exists a write operation op_1 in T_1 and an read operation op_2 in T_2;
2) $T_1 <_H T_2$;
3) $T_1 \rightarrow_W T_2$ does not hold.

Let s_1 be the set of data written by op_1. Assume that transactions are executed according to another transaction history H':..., T_2, ... (where transaction T_1 is removed from transaction history H). Let s_2 be the set of data read by op_2 in transaction history H'. If $s_1 \cap s_2 \neq \emptyset$, we say T_2 is Phantoms Dependent upon T_1, and operations op_1 and op_2 are Phantoms Conflict operations.

We use the notation $\rightarrow_P$ to denote the Phantoms Dependency. Transaction T_j being phantoms dependent upon transaction T_i is denoted by $T_i \rightarrow_P T_j$.

Pseudo-Identity Dependency

Definition 2 **Pseudo-Identity Dependency**. Given a transaction history H:... , T_1 ,..., T_2 ,... and two conflict transactions T_1 and T_2 that satisfy:
1) $T_1 <_H T_1$;
2) there exist a DELETE operation op_1 *[DELETE, x, b_img, -, T_1]* in T_1 and an INSERT operation op_2 *[INSERT, x, -, a_img, T_2]* in T_2, where *x.column* is the PRIMARY KEY or UNIQUE KEY of *x.table*.
Then we say transaction T_2 is Pseudo-Identity Dependent upon T_1, and operations op_1 and op_2 are Pseudo-Identity conflict operations.

We use the notation $\rightarrow_I$ to denote the Pseudo-Identity Dependency. Transaction T_j being pseudo-identity dependent upon transaction T_i is denoted by $T_i \rightarrow_I T_j$.

Domain-Integrity Dependency

Definition 3 **Domain-Integrity Dependency**. Given a transaction history $H: \ldots, T_1, \ldots, T_2, \ldots$ and two transactions T_1 and T_2 ($T_1 <_H T_2$) in H that satisfy:
1) there exist INSERT operations op_1 $[INSERT, x, -, v_1, T_1]$ and op_2 $[INSERT, y, -, v_2, T_2]$ satisfying that $x.v\_pk = y.v\_pk$, and $x.table = y.table$;
2) there exists a row-level domain integrity constraint $CHECK[col_1, col_2, \ldots, col_n]^2$ ($n \geq 2$) on $x.table$ and $x.column$, $y.column \in \{col_1, col_2, \ldots, col_n\}$.
we say transaction T_2 is Domain-Integrity Dependent upon T_1, and operations op_1 and op_2 are Domain-Integrity conflict operations.

We use the notation $\rightarrow_D$ to denote the Domain-Integrity Dependency. Transaction T_j being domain-integrity dependent upon transaction T_i is denoted by $T_i \rightarrow_D T_j$.

Reference-Integrity Dependency

Definition 4 **Reference-Integrity Dependency**. Consider a transaction history H:..., T_1, ..., T_2,... and two transactions T_1 and T_2 ($T_1 <_H T_2$) in H that satisfy:
1) there exist INSERT operations op_1 *[INSERT, x, -, v_1, T_1]* and op_2 *[INSERT, y, -, v_2, T_2]* and
2) there exists a reference integrity constraint from *y.table (y.column)* to *x.table (x.column)* so that the insertion of v_2 is referenced to the value of v_1.
We say transaction T_2 is Reference-Integrity Dependent upon T_1, and operations op_1 and op_2 are Reference-Integrity conflict operations.

We use the notation $\rightarrow_R$ to denote the Reference-Integrity Dependency. Transaction T_j being reference-integrity dependent upon transaction T_i is denoted by $T_i \rightarrow_R T_j$.

Extended Damage Assessment Model

Definition 5 **Transaction Dependency Relation for a Transaction History**. Given a transaction history H, and a binary relations $D = \{< T_i, T_j > | T_i \rightarrow_W T_j$, or $T_i \rightarrow_P T_j$,or $T_i \rightarrow_I T_j$, or $T_i \rightarrow_D T_j$, or $T_i \rightarrow_R T_j\}$ in H, the transaction dependency relation D_H in history H is defined to satisfy D_H=$t(D)$ (here $t(D)$ represents the transitive closure of relation D).

Notation "$\rightarrow$" is introduced to denote the transaction dependency between two transactions. Let $T_i \rightarrow T_j$ denote $< T_i, T_j > \in D_H$.

In a transaction history H: $B \cup G$, where B is the set of malicious transactions and $G = \neg B$. The damage assessment discovers the set of affected transactions A according to following recursive definition:

1) if $< B_k, T_i > \in D_H$, where D_H is the transaction dependency relation in H, then $T_i \in A$;

2) if $T_i \in A$ and $< T_i, T_j > \in D_H$, then $T_j \in A$.

Conclusions and future work

Conclusion and Future Work

- In this paper, we:
  - Analyze the inter-transaction dependencies.
  - Propose four dependency relationships which may cause damage propagation.
  - Give the formal definition of four dependencies.
- We are planning to:
  - Build a damage assessment and recovery prototype based on our model by revamping the kernel of *Dameng* database system.
  - Evaluate performance overhead and compare the results with existing platform.

Thanks ☺

Q & A