CS5570
Architecture of Database Management Systems

Vijay Kumar
Computer Science Electrical Engineering
University of Missouri-Kansas City
Kansas City, MO, USA.

Syllabus

This course deals mainly with the database systems architecture, data warehousing, query optimization and workflow. The entire course includes lectures, project, tests, research report, and a seminar presentation.

Prerequisite: CS431 (Operating System) and CS470 (Introduction to Database Systems).

Textbook: Concurrency Control and Recovery in Database Systems by Bernstein, Hadzilacos, and Goodman available at http://www.csee.umkc.edu/~kumarv/cs570/text_book/complete-CCM-book.pdf

Reference material

A list of "must read" research papers will be provided in the class.
Database Recovery by Vijay Kumar and Sang Song. Kluwer International.
Database Operating Systems, Jim Grey in, An Advanced Course. Ed. by Bayer, Graham, and Seegmuller. Springer-Verlag.
Advanced Transaction Models. Eds. Ahmed Elmagarmid. Morgan and Kauffmann.

Syllabus

Class meets on Mons and Weds at 5:30 to 6:45 PM in Room: FH 306. Ph: 235 2366. Office Hrs: 11:00-12:00PM on MW (Office: RFH 550J).

Chapters	Topics
1	All sections.
2	All sections. At the end of this chapter students are expected to have a good knowledge of serializability.
3	All sections. At the end of this chapter students are expected to have a good insight into concurrency control, especially two-phase locking schemes.
4	All sections, except topics on distributed database systems such as distributed concurrency control.
5	Most sections, excluding distributed database topics.
6	All sections.
	Data Warehousing, Workflow and Query Processing and Optimization (Lecture notes)

Group Projects (Group size 5)

A project is composed of (a) research work, (b) writing a research report, and (c) presenting a seminar on the research topic.

Research

A group should begin their work in the assigned/selected project as early as possible. This will give them enough time to complete their project. The work involves literature survey, reading research papers, thinking about the solution, and so on.

Research report

The format of this report will be explained in the class. This report must be completed and submitted on time.

Seminar

Near the end of semester, each group will present its work. The time duration for a seminar will be 1 hr. and 15 mins.

Tests and Homework

Homework: May be two.

Tests: Two: Midterm and Final.

Points Distribution (flexible): Total Points: 100.

Report:	25.
Seminar:	15.
Homework:	10
Midterm:	20.
Final:	30

Grading Range (These ranges may be revised)

A: (95-100). A-: (90-94). B+: (85-89). B: (80-84). B-: (75-79). C: (70-74).

Outline

- The Basics
- Introduction to Transaction
- Transaction Properties
- Atomicity and Two-Phase Commit
- Availability
- Performance
- Styles of System

Transaction

A transaction is a mechanism for manipulating a database in *consistency-preserving* manner.

Some real-life examples of a transaction

- **Airline seat reservation transaction to buy an airline ticket**
- **On-line purchase from Internet**
- **Withdraw money from an ATM.**
- **E-bay bidding**
- **Etc.**

Transaction Processing (TP)

The *consistency-preserving* requirement of a transaction makes its processing quite hard. It must satisfy the following:

- **Reliability and Availability: system should rarely fail and running all the time**
- **Response and Throughput: within 1 second and at least thousands of transactions/second**
- **Scalability: can grow from balance enquiry to internet scale**
- **Security: must not compromise the institution's integrity**
- **Atomicity: must install its updates in the database successfully**
- **Durability: a transaction is a legal contract, therefore, its updates must persist in the database**

Transaction Processing (TP)

What makes TP Important:

- It is at the core of electronic commerce and soon will be the core of mobile commerce too
- It is the core of most medium-to-large businesses. They use TP for their production systems and cannot operate without it
- It is probably the single largest computer application with more than $80B/year cost
- Nearly all academic research and development use TP to manage their research and data

TP System (DBMS) Infrastructure

End User's Viewpoint

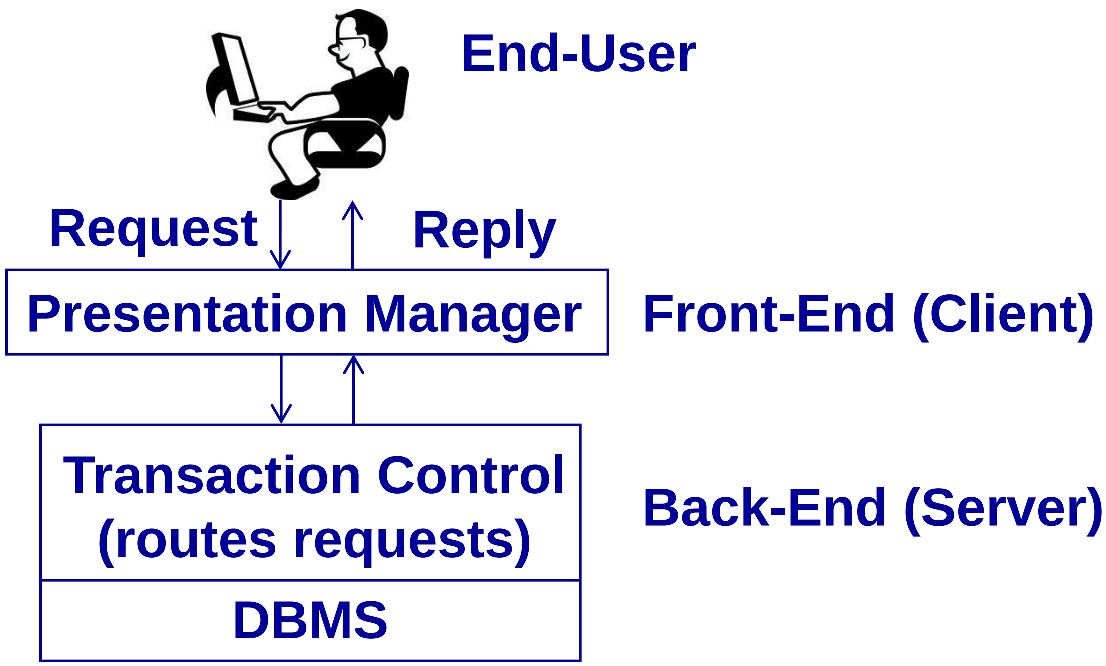
- Enter a request transaction from an input device (monitor, cell phone, PDA, tablet, ATM, browser, etc.)
- The transaction is processed by the DBMS that installs transaction's updates in the database successfully
- The user receives a reply, if a reply is required

The TP system (DBMS) ensures that each transaction

- is an *Atomic* action (independent unit of work),
- executes and *commits* exactly once, and
- produces results that persist (durable) in the database.

TP system has *tools* to enforce these requirements

TP System (DBMS) Infrastructure



DBMS Characteristics

Transaction characteristics

- Typically < 100 transaction types per application (finance, bank, load, etc.)
- Transaction size varies with application. Typically 0-30 disk accesses, 10K - 1M instructions executed, 2-20 messages

A large-scale example: airline reservation system

- 150,000+ active display devices (direct access)
- Indirect DB accesses through Internet (travel agents, customers, etc.)
- thousands of disk drives
- 3000 transactions per second, peak

Application Servers

An application server is a software module that creates, executes and manage TP application. It is also referred to as *TP monitors.* Actually application server can be defined as

App Server = TP monitor + web functionality

Application programmer writes an application (e.g., a transaction) to process a single request and application server scales it up and deploys it on large systems. For example,

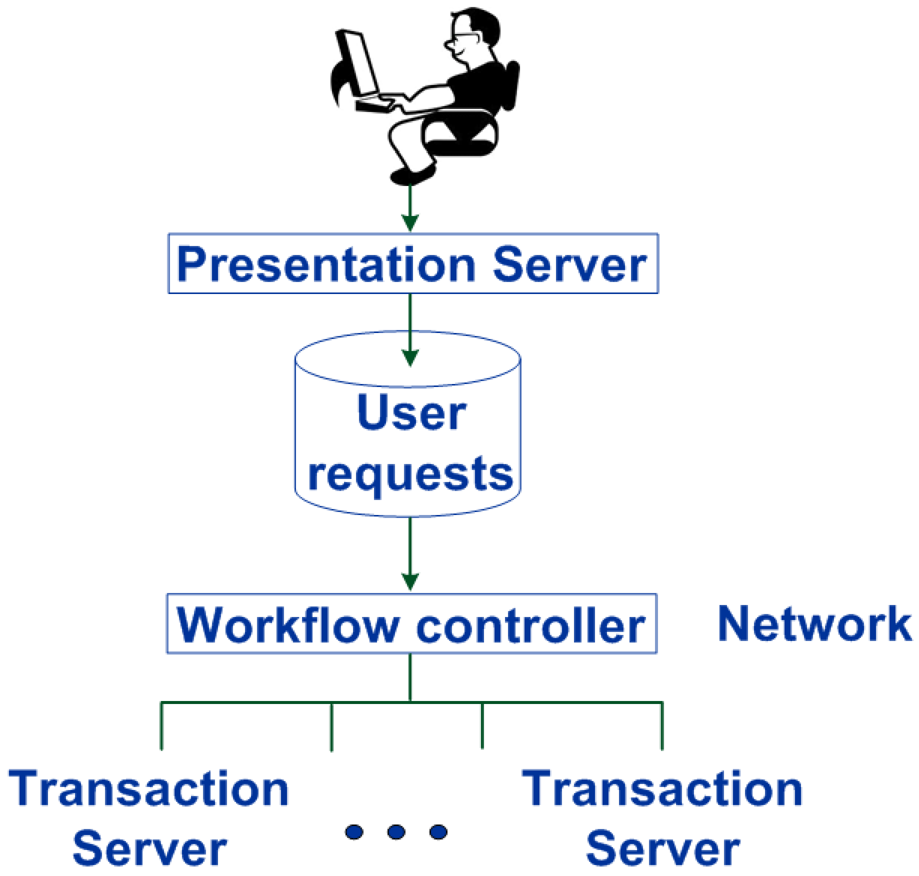
> ***Application developer writes a transaction for debit/credit task. The application server deploys it to 10s/100s of servers and on the Internet.***

Application Servers

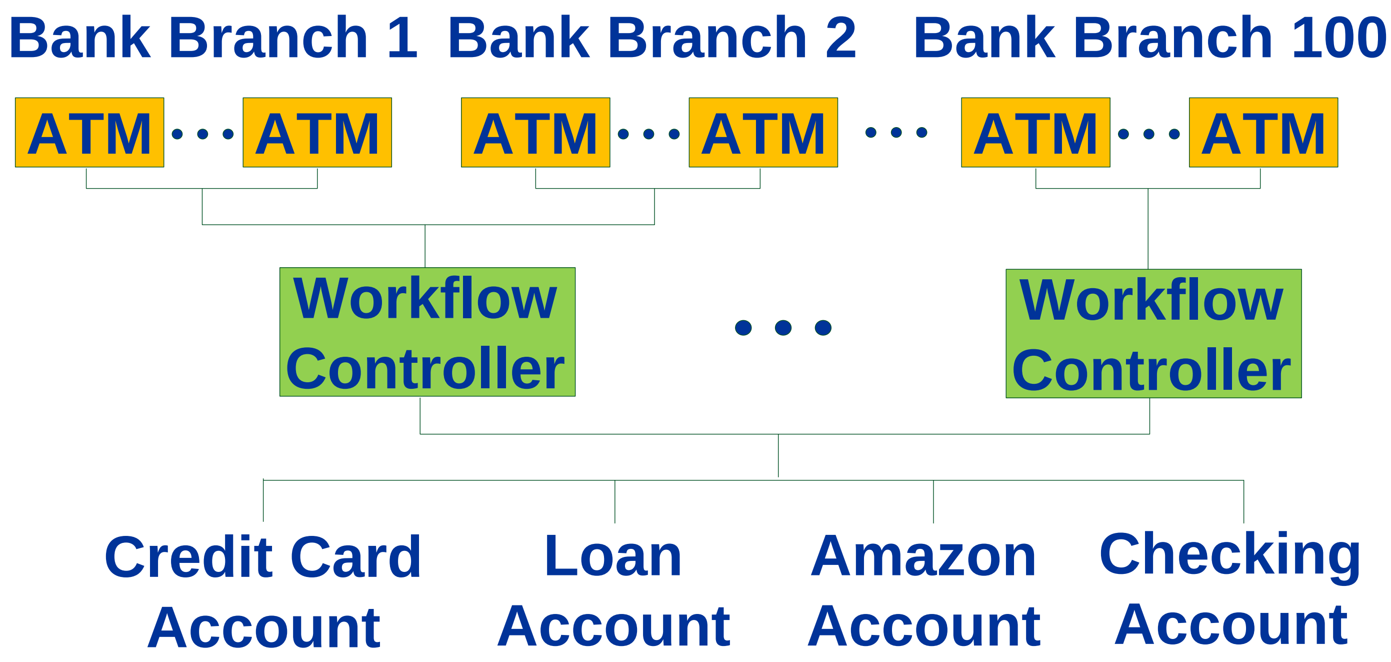
Main components of an application server

- an application programming interface (API) (e.g., Enterprise Java Beans)
- tools for program development (programming language, library, functions, etc.)
- tools for system management (application deployment, auditing, fault and performance monitoring, billing, resource and user management)

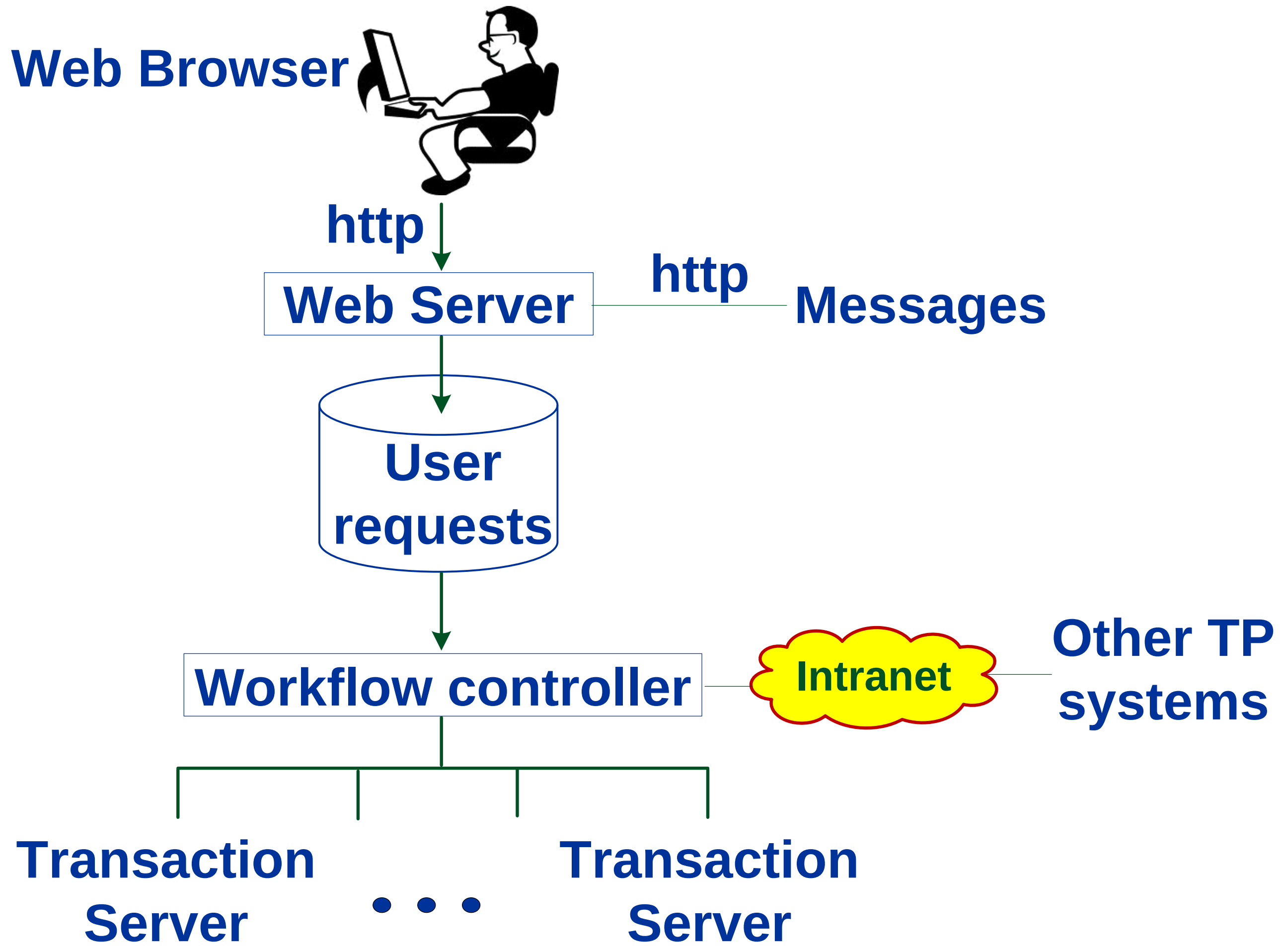
Application Servers Architecture



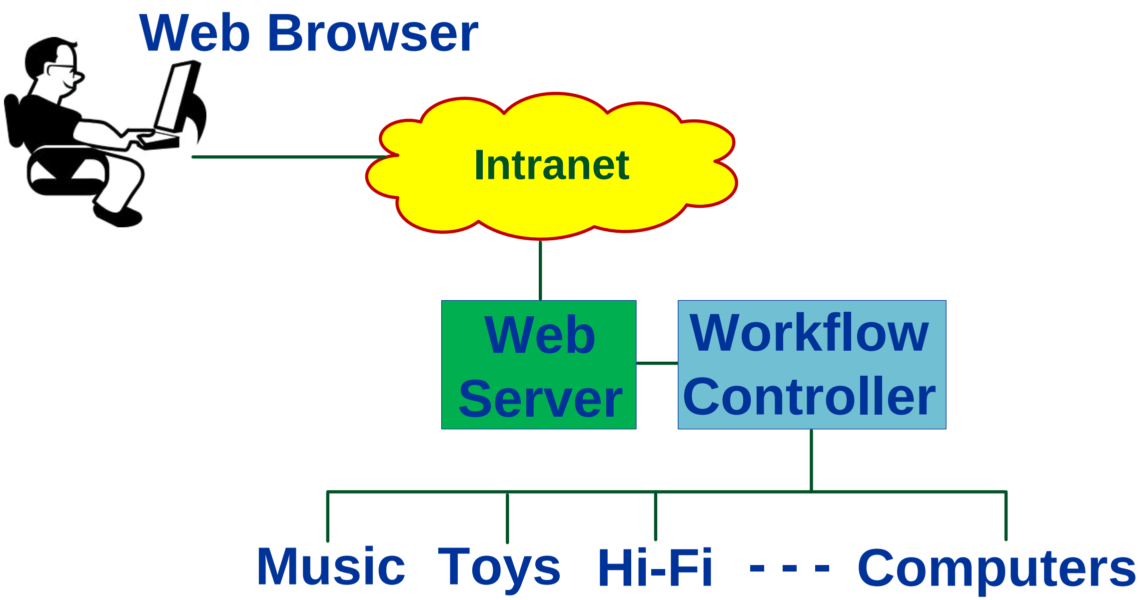
ATM Application Example



Application Servers Architecture (Web)



Internet Retailer



Prologue

We presented an intuitive introduction to the basic components of database management systems. We will now discuss each of these topics formally in detail and understand their theoretical foundations.

Outline

✓ The Basics
- Introduction to Transaction
- Transaction Properties
- Atomicity and Two-Phase Commit
- Availability
- Performance
- Styles of System

Introduction to Transaction

A transaction is a mechanism for manipulating a database in consistency-preserving manner. It changes a consistent state of the database to the next consistent state.

Let S_i be an initial consistent state of the database D ($D = d_1, d_2, \ldots, d_n$; *where* d_i*'s are data items*) and T_i is a transaction. Thus,

T_i

S_i <$is_1, is_2, \ldots, is_m$> S_j ; where $is_1, is_2, \ldots, is_m$ are the intermediate state of the database, and S_j is the final consistent database state.

Introduction to Transaction

- **The successful state transfer $S_i \rightarrow S_j$ guarantees that the intermediate states (is_1, is_2, …, is_m), that may or may not be consistent, was not be visible (not accessible) to other transaction T_j.**
- **The presence of S_j guarantees that the transition is successful.**
- **The inaccessibility of (is_1, is_2, …, is_m) guarantees that T_i performed its updates without any interference.**
- **The persistence of S_j guarantees that the updates of T_i is durable in the database.**

Introduction to Transaction

Database definition

Database (D) = $\{d_1, d_2, ..., d_n\}$; where d_i's are computable objects such a relational or a tuple of a relation.

Every d_i is associated with a set of States $s_{i1}, s_{i2}, ... s_{i\infty}$. A state s_i is said to be consistent if it satisfies all its consistency constraints. If at time t, s_i's of all d_i's satisfy their consistency constraints then D is said to be consistent.

Introduction to Transaction

Consistency Constraints

A set of *assertions* or *constraints* that must be satisfied by all operations to the database. These constraints may be explicitly defined or implied when transaction code is written.

Examples

Last account balance = Current balance + Debit amount

Cost of an item = Amount paid - Tax

Transaction Properties

The four properties discussed earlier is formally defined as

- **Atomicity:** All or nothing. Transaction either completes successfully or does not have any effect
- **Consistency:** Preserves database integrity
- **I solation:** Entire execution is interference-free
- **Durability:** Changes to the database are not lost by a failure

ACID: If a program possesses ACID then it is a transaction.

Transaction Properties

Atomicity

A transaction either executes *completely* (results persists in the database) or *never starts* (no result is installed in the database). For example, a money transfer transaction (debit-credit transaction) it debits one account and credits the other. Either debit and credit both run, or neither runs.

Successful completion is called *Commit*. The "*never starts*" state is enforced through *Abort where executed operations (e.g., a write) is undone (restore the last consistent value). Commit* and *abort* are irrevocable actions.

But some real world operations are not undoable. Examples - ticket printed, fire missile fired, a hole drilled, etc.

Example - ATM Dispenses Money (a non-undoable operation)

T_1: Start

. . .

Dispense Money

Commit

System crashes
Transaction aborts but
Money is dispensed

T_1 : Start

. . .

Commit

Dispense Money

System crashes

Deferred operation never gets executed

Reading Uncommitted Output Isn't Undoable

T_1*: Start*

. . .

Display output → **User reads output**

. . .

If error, Abort

T_2 *: Start*

Get input from display → **User enters input**

. . .

Commit

Compensating Transactions

A transaction that reverses the effect of a committed transaction.

Example,

- A debit-credit transaction
- Annul a marriage

Certain transactions may not have compensating transaction

Example

- Fire missile
- Drill a hole

A well-designed TP application should have a compensation for every transaction type

Consistency

Consistency *C = {c1, c2, …, cn}*

Database D = {d1, d2, …dn}

Consistent *D*

> ***{<d$_1$ satisfies c$_1$>,***
> ***<d$_2$ satisfies c$_2$ >,***
> ***…***
> ***<d$_n$ satisfies c$_n$ >}***

T —Operations (read, write)—

- $<s_1, d_1, c_1> \rightarrow$ State change $\rightarrow <s'_1, d'_1, c'_1>$
- $<s_2, d_2, c_2> \rightarrow$ State change $\rightarrow <s'_2, d'_2, c'_2>$
- • • •
- $<s_n, d_n, c_n> \rightarrow$ State change $\rightarrow <s'_n, d'_n, c'_n>$

Unlike *A*, *I*, and *D* *consistency preservation is a property of a transaction, not of the TP system*

Some Notations

Data Item

We will denote a data item as x, y, z, etc. A data item can be a relation, a tuple, a file, etc.

Transaction

We will denote transactions as T_1, T_2, …, T_i.

Basic Operations (Indivisible operations)

Read = r. Thus, $r_i[x]$ = Read x by T_i (x value does not change)
Write = w. Thus, $w_i[x]$ = Write x by T_i (x value changes)
Commit = c. Thus, c_i = Commit of T_i
Abort = a. Thus, a_i = T_i is aborted (a system operation)

Formalization of a Transaction

A transaction executes a set of reads and writes to manipulate the database. Some of the operations of a transaction can be executed in parallel if running on a multiprocessor system.

Let T_1 has $r_1[x]\ r_1[y]\ w_1[z]$. Its execution can be represented graphically as:

$$\begin{tikzcd} r_1[x] \arrow[dr] & & \\ & w_1[z] \arrow[r] & c_1 \\ r_1[y] \arrow[ur] & & \end{tikzcd}$$

Since $w_1[z]$ is not a function of any r_1, the execution of these operations in any order will give the same correct result. Thus, all possible schedules are correct. This indicates that a transaction is a partial order.

Formalization of a Transaction

Operations of some transactions could be totally ordered. Consider the following transaction:

T_1 **has** $r_1[x]\ w_1[x]\ r_1[y]\ w_1[y]$

In this schedule each w_1 **is a function of** r_1**. If we assume that the new value of** y **is computed using** x **then this is the only correct order of** r_1 **and** w_1**. Thus, the order is total.**

We will model a transaction as a partial order. In doing so we will not violate the constraints of total order.

Formalization of a Transaction

We formally define a transaction as follows:

A transaction Ti is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$
2. $a_i \in T_i$ iff $c_i \notin T_i$
3. if t is a_i or c_i (whichever is in T_i), for any other operation $p \in T_i$, $p <_i t$; and
4. If $r_i[x], w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$

Interpretation of a Transaction

The formal definition of a transaction does not model its execution. That is, it does not use the initial and final values of a data item because the transaction structure is not related to them. This model works for all values so the execution is "uninterpreted" or "unspecified." However, when we deal with schedulers, we will take into consideration data values.

History

History (H)

A history is a sequence of these operations *(r, w, c, a)* of a set of concurrent transactions *(T_1, T_2, …, T_n)* in the order that the database system processed them. It gives a complete picture of execution of these transactions.

Example of a History

r1[x] r2[x] w1[x] w2[x] r3[y] r4[z] w2[y] w3[y] w4[z] c1 a2 a3 c4

Schedule

Schedule (S)

A schedule is any prefix of H. It may or may not contain a or c. A history can be called a schedule but a schedule may not represent a history.

Examples of a Schedule

$r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ r_3[y]\ r_4[z]\ w_2[y]$ ***or***

$r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ r_3[y]\ r_4[z]\ w_2[y]\ c_1\ a_2$ ***or***

$r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ r_3[y]\ r_4[z]\ w_2[y]\ c_1\ c_2\ c_3\ a_4$ ***(a history also)***

Etc.

History Vs. Schedule

A history (H) is a complete sequence of operations (*r*, *w*, *c*, *a*) for concurrent transactions ($T_1, T_2, \ldots, T_n$) in the order that the database system processed them.

A schedule (S) of ($T_1, T_2, \ldots, T_n$) is any prefix of *H*. The prefix may contain c or a as well.

$H = r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ r_3[y]\ r_4[z]\ w_2[y]\ w_3[y]\ w_4[z]\ c_1\ a_2\ a_3\ c_4$. This can be called a schedule as well.

$S = r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ r_3[y]\ r_4[z]\ w_2[y]$. This is not a history.

A history is a complete event where as a schedule could be an ongoing event.

History

Formalization of a history (H)

Let T = ($T_1, T_2, \ldots, T_n$). A complete H over T is a partial order with ordering relation $<_H$ where:

1. $H = \bigcup_{i=1}^{n} T_i$;
2. $<_H \supseteq \bigcup_{i=1}^{n} <_i$; *and*
3. *For any two conflicting operations p, q either p* $<_H$ *q or q* $<_H$ *p*

Condition 1: H includes precisely the operations executed by ($T_1, T_2, \ldots, T_n$).

Condition 2: Operations of each Ti are honored precisely.

Condition 3: Order of conflicting operations p and q is determined by $<_H$.

Types of History

The execution of a set of $T = (T_1, T_2, \ldots, T_n)$ generate the following types of histories

- **Complete history**
- **Partial history (a prefix of a complete history)**
- **Committed history**

We will use the following transaction to formalize types of histories

$T1 = r_1[x] \rightarrow w_1[x] \rightarrow c_1$

$T3 = r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3$

$T4 = r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_4$

(An $\rightarrow$ defines the order (precedence) of r, w, c, and a operation. We will remove them when there is no ambiguity.)

Types of History

Complete history

A complete (includes complete execution of all committed transaction. It may include operations c or a or both) history over T (T_1, T_3, T_4) is

$$r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3$$

$$H_1 = \quad r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_4$$

$$r_1[x] \rightarrow w_1[x] \rightarrow c_1$$

Since all transactions are committed, this is a committed history also.

Types of History

Partial history (a prefix of a history)

A prefix of a history is a partial history where some transactions may be active (not committed or aborted). A prefix of history (H_1) over T (T_1, T_3, T_4) is

$$
\begin{array}{ll}
 & r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \\
 & \uparrow \qquad\qquad\quad \uparrow \\
H_1 = & r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \\
 & \qquad\quad \uparrow \\
 & r_1[x] \rightarrow w_1[x] \rightarrow c_1
\end{array}
$$

Types of History

Committed history

A committed history includes only committed transaction (no aborted transaction)

A committed history over $T = (T_1, T_3, T_4)$ is

$$H_c = \begin{array}{l} r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3 \\ \uparrow \qquad\qquad \uparrow \\ r_1[x] \rightarrow w_1[x] \rightarrow c_1 \end{array}$$

History discussion

A history may have a total order of operations. This happens only when all its transactions have total order.

$r_1[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3.$

This can also be written as

$r_1[x]\ w_1[x]\ c_1\ r_3[x]\ w_3[y]\ w_3[x]\ c_3.$

T_i is committed in a complete history ($C(H)$) if $c_i \in H$.

T_i is aborted in a complete history ($C(H)$) if $a_i \in H$.

Serializable Histories

Database must preserve consistency before and after the execution of $T = (T_1, T_2, \ldots, T_n)$. Database consistency is guaranteed if all histories over the database are serializable.

A history is seriazable, if and only if *T* produces the same result as produced by a serial execution of *T*. A serial execution of *T* is consistency preserving but suffer with poor resource utilization and low throughput. So $T_1, T_2, \ldots, T_n$ are run concurrently or simultaneously.

Serializable Histories

To see if a history of *T* is serialiazble, it is compared with a serial history of *T*. If serial $H_s(T) \equiv$ concurrent history $H_c(T)$ then $H_c(T)$ is serializable. First we define equivalence criteria for any two histories H and H'. H ≡ H' if

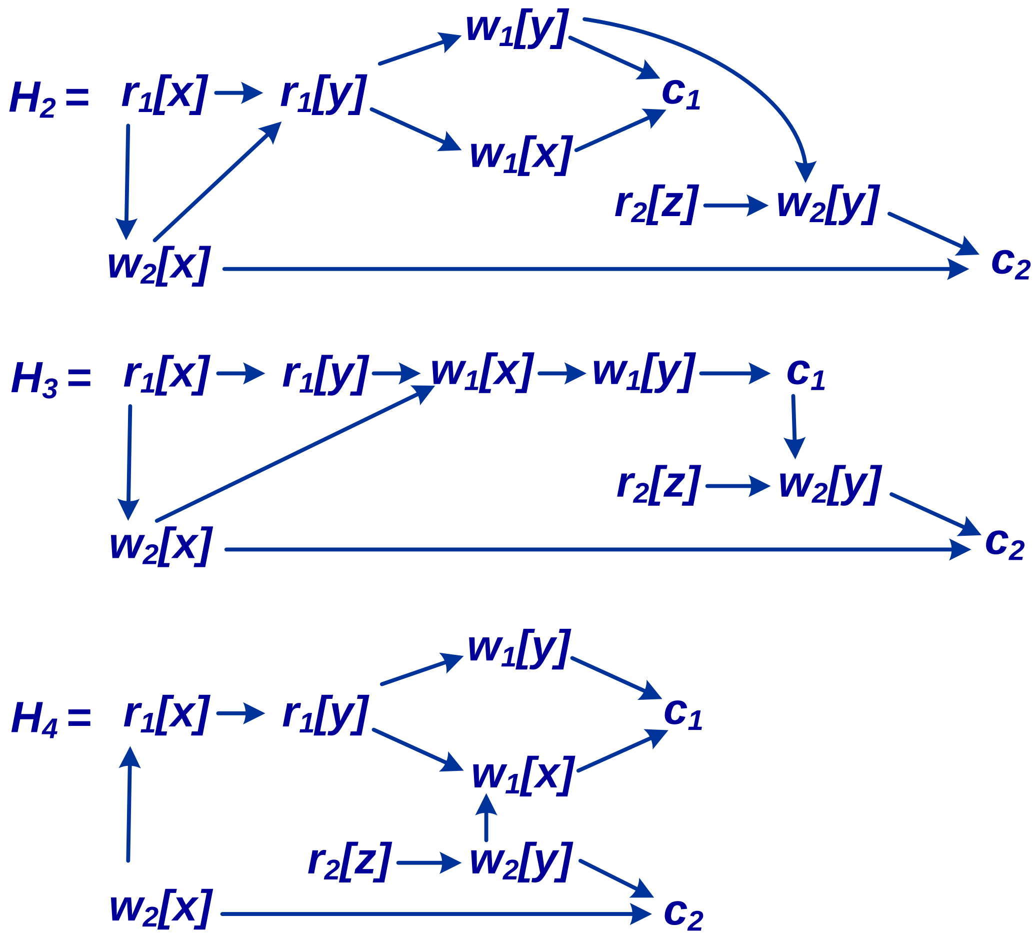
1. **They are defined over the same set of transaction, i.e., over $T = (T_1, T_2, \ldots, T_n)$.**
2. **The order of conflicting operations of active or committed transactions are the same in H and H'. Thus, if $p_i <_H q_j$ then $p_i <_{H'} q_j$ where a_i , $a_j \notin$ H and H'.**

$H = r_3[x]\ w_3[y]\ w_3[x]\ r_1[x]\ w_1[x]\ c_3\ c_1$ ($w_3 < r_1$ and $w_3 < w_1$)
$H' = r_3[x]\ w_3[y]\ r_1[x]\ w_1[x]\ w_3[x]\ c_3\ c_1$ ($r_1 < w_3$ and $w_1 < w_3$)

H is not equivalent to H'.

Serializable Histories

$H_2 \equiv H_3?$ $H_2 \equiv H_4?$ $H_3 \equiv H_4?$



Serializable Histories

There may be more than one complete H_c of T. However, each H_s is unique. What we require is that at least one of the H_c's should be equivalent to a H_s. We face a problem with comparing an H_s with one of the H_c's because the H_c contains active transactions as well and the fate of active transactions cannot be predicted. So comparison is done with the committed project of H_c.

Serializable Histories

Consider the following history

$$H_i = r_1[x]\ r_3[x]\ w_3[y]\ w_3[x]\ r_4[x]\ w_1[x]\ w_4[x]\ c_3\ c_4$$

T_1 started first and active. T_3 committed (it is incorrect but it is not important here). The serial serial schedule should be T_1 then T_3. But we cannot compare it with any H_c because the fate of T_1 is unpredictable. If we take the committed project, i.e., $r_3[x]\ w_3[y]\ w_3[x]\ r_4[x]\ w_4[x]\ c_3\ c_4$ of H_i we can see that it is equivalent to a serial history T_3 then T_4.

Serializable Histories

Question and Answer

The Serializability Theorem

We can now discuss the serializability theorem that provides us a criteria for discovering if a schedule is serializable. First we introduce "Serializability Graph (SG)." A SG for a H, SG(H), is a DAG. Its nodes are the transactions that are committed in H. The edge $T_i \rightarrow T_j$ The indicates that all conflicting operations of T_i come before all conflicting operations of T_j. Consider

$$
\begin{tikzcd}
 & r_3[x] \arrow[r] & w_3[x] \arrow[r] & c_3 \\
H_5 = r_1[x] \arrow[r] & w_1[x] \arrow[r] \arrow[ur] & w_1[y] \arrow[r] & c_1 \\
r_2[x] \arrow[r] \arrow[ur] & w_2[y] \arrow[r] \arrow[ur] & c_2 &
\end{tikzcd}
\qquad
SG(H_5) = \begin{tikzcd} T2 \arrow[r] \arrow[rr, bend left] & T1 \arrow[r] & T3 \end{tikzcd}
$$

If $w_3[x]$ is replaced by $w_3[z]$ then $SG(H_5)$ becomes $T2 \rightarrow T1 \rightarrow T3$.

The Serializability Theorem (2.1)

SG(H₅) **is acyclic. Now consider the history H6:**

$$H_6 = r_1[x]\ w_1[x]\ r_3[x]\ w_3[x]\ r_3[y]\ w_3[y]\ r_1[y]\ w_1[y]\ c_1\ c_3$$

SG(H₆) **is cyclic because it is**

$$SG(H_6) = \begin{tikzcd} T1 \arrow[r, bend left] & T3 \arrow[l, bend left] \end{tikzcd}$$

For this reason it is not equivalent to any serial history (there is no serial history of *H_6*). Thus, the serializability theorem state:

A history (H) is seriazable iff SG(H) is acyclic.

Proof of Serializability Theorem

***A history (H) is seriazable iff SG(H) is acyclic*.**

Let $T = \{T_1, T_2, \ldots, T_n\}$. Let $C(H)$ be a history over $\{T_1, T_2, \ldots, T_m\}$ $(m \leq n)$ which are nodes of $SG(H)$. If $SG(H)$ is acyclic then its nodes may be topologically sorted. Let one of the topological sorts of $SG(H)$ is $T_{i_1}, T_{i_2}, \ldots, T_{i_m}$. Let H_s is a serial history of $\{T_1, T_2, \ldots, T_m\}$. We claim that $C(H) \equiv H_s$.

Why?

Let $p_i \in T_i$ (committed in H) and $q_j \in T_j$ (committed in H) be two conflicting operations. If $p_i <_H q_j$ then $SG(H) \in T_i \rightarrow T_j$ (by definition). Therefore, in any topological sort of $SG(H)$, $T_i \rightarrow T_j$ will persists. This implies that all conflicting operations of T_i appear before all conflicting operations of T_j. In any serial execution of T_i *and* T_j this order will persist which means there cannot be a cycle in $SG(H)$.

Proof of Serializability Theorem

Confirmation by contradiction

Suppose there is a cycle is *SG(H)*. This means at some place in the history we must have $T_i \rightarrow T_j \rightarrow T_i$. Let the cycle be $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$. This implies that in *SG(H$_s$)* we have $T_1 \rightarrow T_1$. This is not possible. That is *SG(H)* is an acyclic graph. □

Serializability Theorem

An acyclic *SG(H)* may have more than one topological sort. In this situation it is possible that *H* may be equivalent to more than one serial history. Consider the following history

$$H_6 = w_1[x]\ w_1[y]\ c_1\ r_2[x]\ r_3[y]\ w_2[x]\ c_2\ w_3[y]\ c_3$$

$$\begin{tikzcd} SG(H_6) = T_1 \arrow[r] \arrow[rr, bend left] & T_3 & T_2 \end{tikzcd}$$

SG(H₆) **has two topological sorts:** $T_1 \rightarrow T_2 \rightarrow T_3$ **and** $T_1 \rightarrow T_3 \rightarrow T_2$ **.** H_6 **is equivalent to both.**

Recoverable Histories

A history defines the state of the database. A history can clearly indicate if the database can be recovered or not. It also indicates how many transactions have to be rolled-back or rolled-forward to recover the database. We, therefore, need to define recoverable and non-recoverable history. Let us first understand "*reads from*" relationship.

Recoverable Histories

"Reads From"

Let T_j is an active transaction and it wrote into data item x. T_i reads x while T_j is still active. We say T_i reads from T_j in H. If T_i reads x after T_j has aborted then T_i does not read from T_j. Formally,

1. $w_j[x] < r_i[x]$;
2. $a_j \nless r_i[x]$ *and*
3. *If there is some* $w_k[x]$ *such that* $w_j[x] < w_k[x] < r_i[x]$, *then* $a_k < r_i[x]$.

Recoverable History

A history H is recoverable (*RC*) if, whenever T_i reads from T_j $(i \neq j)$ in *H* and $c_i \in H$, $c_j < c_i$. Intuitively, if a dependent transaction (a transaction that reads from another transaction) in *H* commits after the transaction from which it reads then *H* is recoverable. This means that the database can recover from a failure. In this example T_j is dependent on T_i so T_j commits after T_i does in *H*.

Avoids Cascading Abort (ACA) History

A history *H* is *ACA* if, whenever T_i reads from T_j $(i \neq j)$ in *H* and $c_j < r_i[x]$. This means that in *ACA* a transaction reads those values that are written by a committed transaction. Note that this is not a read from relation because the transaction is committed. Also T_i is not dependent on T_j because T_j is committed before T_i reads its committed value.

Note: It involves only read operation

Strict (ST) History

A history *H* is *ST* if, whenever $w_j[x] < o_i[x]$ $(i \neq j)$, *either* $a_j < o_i[x]$ *or* $c_j < o_i[x]$ *where* $o_i[x]$ *is* $r_i[x]$ *or* $w_i[x]$. ***ACA includes only read operation but ST includes both read and write.***

This means that T_i cannot read or write to *x* until T_j is committed or aborted.

Examples of RC, ACA, and ST History

$T_1 = w_1[x]\ w_1[y]\ w_1[z]\ c_1$

$T_2 = r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$

Histories

- $H_7 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2\ w_1[z]\ c_1$
- $H_8 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ w_1[z]\ c_1\ c_2$
- $H_9 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ w_1[z]\ c_1\ r_2[y]\ w_2[y]\ c_2$
- $H_{10} = w_1[x]\ w_1[y]\ r_2[u]\ w_1[z]\ c_1\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$

H_7 = *Not RC* ($c_2 < c_1$ *and* T_2 *reads from* T_1)

H_8 = *RC* ($c_1 < c_2$ *and* T_2 *reads from* T_1)

H_9 = *ACA* ($c_1 < c_2$ *and* T_2 *reads after* c_1)

H_{10} = *ST* ($c_1 < c_2$ *and* T_2 *reads and writes after* c_1)

Examples of RC, ACA, and ST History

Question and Answer

Theorem ST $\subset$ ACA $\subset$ RC (2.2)

ST is more restricted than ACA and ACA is more restricted than RC.

Proof: Let H $\in$ ST. This means Ti reads and writes after T_j has applied its updates and committed. If T_i reads x from T_j in H ($i \neq j$) then $w_j[x] < r_i[x]$, $a_j \nless r_i[x]$ and by definition of ST, $c_j < r_i[x]$. This satisfies ACA criteria. Therefore H $\subseteq$ ACA. History H, (above) avoids cascading aborts but is not strict, implying $ST \neq ACA$. Hence ST $\subset$ ACA.

Now let $H \in ACA$. This means T_i reads x from T_j in H and $c_j \in H$. Since H is ACA it must must have $w_j[x] < c_j < r_i[x]$. Since $c_i \in$ H, $r_i[x] < c_i$ and therefore $c_j < c_i$, proving H $\in$ RC. Thus ACA $\subseteq$ RC. History H_8, (above) is in RC but not in ACA, proving ACA $\neq$ RC. Hence ACA $\subset$ RC.

Prefix Commit-Closed Property

Prefix Commit-Closed is a property of a history. Let *H* be a history and *C(H')* is any prefix of *H*.

If the property is true for H then it is also true for C(H').

Explanation: If *H* is a correct history (produces a consistent state, i.e., produced by a correct scheduler) then any prefix *H'* can also be produced by the same scheduler. Since the scheduler is correct, *H'* should also be correct.

$H = w_1[x]\ w_1[y]\ c_1\ r_2[x]\ r_3[y]\ w_2[x]\ c_2\ w_3[y]\ c_3$

$H' = w_1[x]\ w_1[y]\ c_1\ r_2[x]\ r_3[y]$ ***System failed.***

The recovery module will leave T_1's updates in the database but will remove partial updates of T_2 and T_3 to maintain atomicity.

Prefix Commit-Closed Theorem (2.3)

If *H* is an *SR (serializable)* then for any prefix *C(H')* of *H* is also *SR*.

Proof (refer to theorem 2.1): If H is an SR then its graph SG(H) will be acyclic. Let C(H') is a prefix of H. If the edge $T_i \rightarrow T_j$ exist in SG(H) then it will also exist in SG(C(H')). This means that all conflicting operation of T_i will come before all conflicting operations of T_j in both graphs. In particular, if conflicting operations $p_i <_H p_j$ appear in H then $p_i <_{H'} p_j$ will appear in C(H'). Since SG(H) is acyclic, SG(C(H')) will be acyclic too.

Increment and Decrement Operations

***Increment (I)*: increase the data value by 1.**

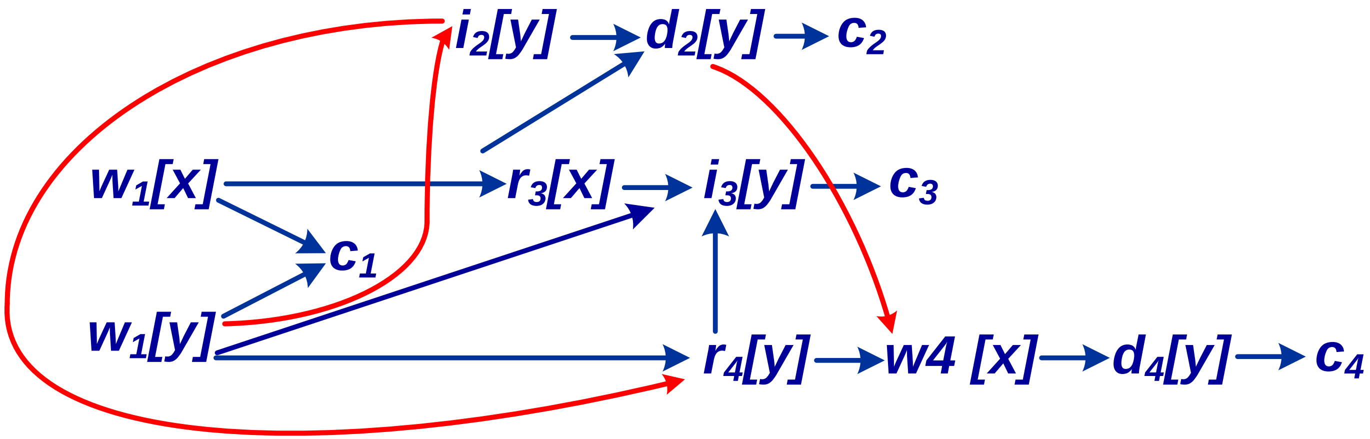
***Decrement (D):* decrease the value by 1.**

Unlike write (it could be a function of read) *I* and *D* always commute and they do not return a result. We create a compatibility matrix that adds *I* and *D* to *r* and *w* operations.

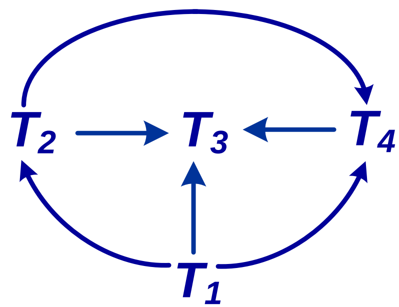
	Read	Write	Increment	Decrement
Read	Y	N	N	N
Write	N	N	N	N
Increment	N	N	Y	Y
Decrement	N	N	Y	Y

Increment and Decrement Operations

$SG(H_{11})$ =



SG(H₁₁) =



SG(H11) is acyclic and is equivalent to a serial history T1→T3→T2→T4 that can be obtained from a topological sort.

Increment and Decrement Operations

Question and Answer

View

In database a view is a part of the whole. The whole can be a relation, a file, or a history. Here a view is used to analyze the history equivalence using the final results produced by two histories. In our earlier treatment of history equivalence, we used conflicting operation. In the view approach we look at the final writes in two histories to see if they produce the same result. Note that when we analyze a write, we must consider read operations that precede the write.

View Equivalence

We proceed as follows:

We assume that each write is a function of read. We can formalize this as follows:

1. **If each transaction reads each of its data item from the same *W's* in *H* and *H'* then all *W's* will produce the same result in *H* and *H'***
2. **If for each data item *(x)*, the *w(x)* is the same in *H* and *H'* then the final value of all data items will be the same in *H* and *H'***

We conclude that if all *W's* in *H* and *H'* write the same final values then this will leave the database in the same consistent state.

Formalization of View Equivalence

The final write of x in H is $w_i[x] \in H$ and $a_i \notin H$. For any $w_j[x] \in H$ ($j \neq i$) either $w_i[x] < w_j[x]$ or $a_i \in H$. H and H' are equivalent if

1. **$H \in \{T_1, T_2, \ldots, T_n\}$ and $H' \in \{T_1, T_2, \ldots, T_n\}$ and have the same set of operations;**
2. **For any T_i, T_j such that $a_i, a_j \notin H$ (hence $a_i, a_j \notin$ H') and for any x, if T_i reads x from T_j in H then T_i reads x from T_j in H' and**
3. **For each x, if $w_i(x)$ is the final write in H then it is also the final write in H'.**

View Serializability

The final write of x in H is $w_i[x] \in H$ and $a_i \notin H$. For any $w_j[x] \in H$ ($j \neq i$) either $w_i[x] < w_j[x]$ or $a_j \in H$. H and H' are equivalent if

1. **$H \in \{T_1, T_2, \ldots, T_n\}$ and $H' \in \{T_1, T_2, \ldots, T_n\}$ and have the same set of operations;**
2. **For any T_i, T_j such that $a_i, a_j \notin H$ (hence $a_i, a_j \notin$ H') and for any x, if T_i reads x from T_j in H then T_i reads x from T_j in H' and**
3. **For each x, if $w_i(x)$ is the final write in H then it is also the final write in H'.**

View Serializability

Definition: A history H is view serializable (VSR) if for any prefix $H' = C(H')$ (committed projection) of H is view equivalent to some serial history. Consider the following history:

$$H_{12} = w_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ c_1\ w_3[x]\ w_3[y]\ c_3$$

$C(H_{12}) = H_{12}$ is a view equivalent to $T_1\ T_2\ T_3$.

$C(H'_{12}) = w_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ c_1$ is not view equivalent to either $T_1\ T_2$ or $T_2\ T_1$.

Note: A committed projection of H is $w_3[x]\ w_3[y]\ c_3$

A prefix $C(H')$ of H is $w_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ c_1$

View Serializability

Self Study: Theorem 2.4 (Page 40)

Question and Answer

End of History and Transaction structure.

Next topic: Chapter 1
Serializability, Recoverability and DBMS structure