ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Snapshot Isolation

Christian Plattner, Gustavo Alonso
Exercises for Verteilte Systeme WS05/06
Swiss Federal Institute of Technology (ETH), Zürich
{plattner,alonso}@inf.ethz.ch

Today

- Reminder: Traditional Concurrency Control in Databases
- Extended ANSI SQL Isolation Levels
- Snapshot Isolation (SI)
- Implementation of SI in Real Databases
- How to deal with SI

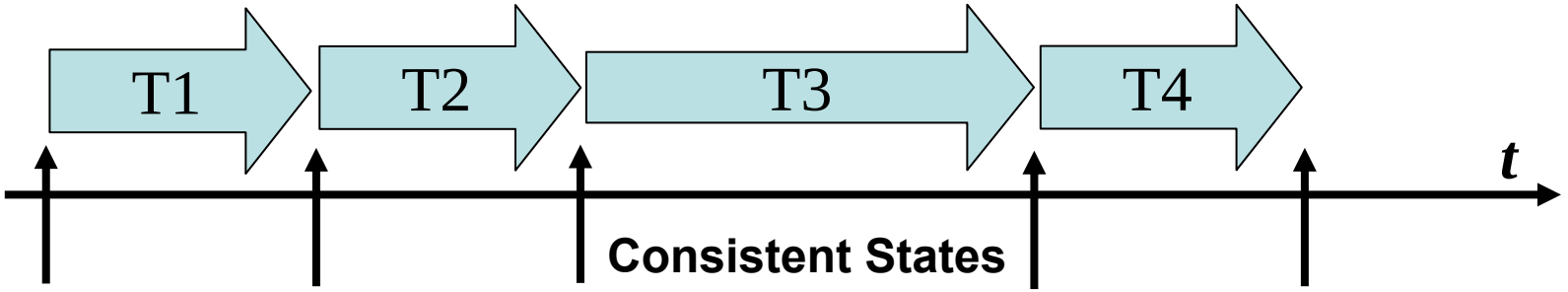
- Introduction to the Mandatory Exercise 3

Transaction Processing in Databases

- Databases execute transactions according to the ACID paradigma (Atomicity, Consistency, Isolation, Durability).
- Each transaction ***i*** starts with a begin $\mathbf{b_i}$ and then continues with a (possible empty) partially ordered sequence of read $\mathbf{r_i(X)}$ and write $\mathbf{w_i(X)}$ operations (X and Y denote arbitrary data elements). Transactions terminate either with an abort $\mathbf{a_i}$ or commit $\mathbf{c_i}$ operation.
- A transaction that terminates with an *abort* does not lead to any changes in the database (→***atomicity***, all or nothing). If a transaction *commits*, then all its changes have to be stored persistently (→ ***durability***).
- However, from the user‘s perspective, each transaction consists of SQL statements (e.g., `BEGIN, SELECT, INSERT, UPDATE, DELETE, COMMIT, ROLLBACK`). These high level operations are automatically mapped to the above described elementary operations.

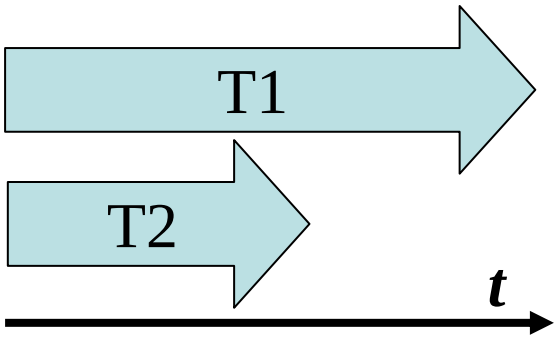
Consistency and Isolation

- Each committed transaction moves the database from a consistent state to the next consistent state.



- As long as the database executes transactions in a ***serial*** fashion, then transaction ***isolation*** is automatically guaranteed.
- However, obviously, that is not very efficient as the resources dedicated to the database can typically not be fully used in this way.
- We therefore want to be able to execute transactions in parallel, without violating the ACID guarantees of the database.

Executing Transactions in Parallel



T1: X := X + 100; Y := Y + 100;

T2: X := X * 2

$b_1\ b_2\ r_1(X)\ r_2(X)\ w_2(X)\ c_2\ w_2(X)\ r_2(Y)\ w_2(Y)\ c_2$

- Blindly executing transactions in parallel can yield unexpected results (the above shown schedule of operations suffers from the so called ***lost update*** problem).
- The lost update phenomenon occurs whenever two transactions, while attempting to modify a data item, both read the item's old value before either of them writes the item's new value. This is just one example of what can go wrong – there are many other possiblities (e.g., ***inconsistent retrieval)***.
- Why did things go wrong? Intuitive idea: no serial execution of the two transactions (either T1,T2 or T2,T1) would lead to the same result (e.g., the *observed* and *written* values by the transactions) → the schedule is ***not serialisable***.

Concurrency Control based on Conflicts

- We need a concurrency control component in the database which assures that any concurrent execution of transactions leads to a schedule that is *somehow* equivalent to a serial execution. But how to define equivalence?
- If we closely inspect the interactions between the operations of different transactions, then we can observe that operations on the same data item may ***conflict***.
- Two operations are defined to conflict if, in general, the computational effect of their execution depends on the order in which they are processed. The computational effect of the two operations consists of both the value returned by each operation (if any) and the final value of the data item(s) they access.
- $\mathbf{r_1(X)\ r_2(X)}$ → no conflict, $\mathbf{r_1(X)\ w_2(X)}$ → conflict!, $\mathbf{w_1(X)\ w_2(X)}$ → conflict!
- A schedule is ***conflict serialisable***, if it orders conflicting operations of committed transactions equally to some serial schedule.

Enforcing Conflict Serialisability

- Idea: ***delay*** certain incoming operations so that the resulting schedule is conflict serialisable.
- Is that enough for real world applications? No, furthermore, e.g., we would like to ***avoid cascading aborts*** (transactions that have to be aborted because they read values produced by a concurrent transaction that aborts, hence they rely on values that never existed in the database).
- How to achieve this? → Use two phase locking (***2PL***). Each operation has to obtain a lock first. Only when the lock is granted, the operation may be executed. In case of deadlocks one must abort transactions (as many as needed).
- Normally, implementations use ***strict*** 2PL: the locks of a transaction T_i are all released together (***after*** the execution of either $\boldsymbol{c_i}$ or $\boldsymbol{a_i}$).

Improving Concurrency

- If we only use one kind of lock, then concurrency may be lowered: e.g., different readers of the same element may block each other.
- To improve concurrency, one typically defines ***different types of locks*** (read and write locks correlating to the attempted operations) and a ***compatibility matrix***.
- Still, things are not optimal: for every data element touched in the database by a transaction a lock has to be aquired. E.g., a sequentical scan on a huge table may lead to a lock request for each tuple. Solution: use ***dynamic lock escalation*** by locking elements on a higher level (e.g., lock disk blocks or the full table). This leads to less locking operations. Big disadvantage: we now may lock too many elements and concurrency is lowered.
- Other solution: allow different (lower) degrees of isolation for transactions that ask for it.

Extended ANSI SQL Isolation Levels

Based on “Phenomenas” P0-P3:

- **P0: Dirty Write** → It is possible to update a value that was already updated by a concurrent, uncommitted transaction.
- **P1: Dirty Read** → Reading a value that was updated by a concurrent, uncommitted transaction.
- **P2: Fuzzy Read** (non-repeatable read) → Reading a value twice gives different results, because of a concurrent update inbetween.
- **P3: Phantom Read** → Using the same selection criteria on a table twice gives different sets of results: a concurrent updater deleted or inserted elements.

Extended ANSI SQL Isolation Levels

Table 3. ANSI SQL Isolation Levels Defined in terms of the four phenomena

Isolation Level	P 0 Dirty Write	P 1 Dirty Read	P 2 Fuzzy Read	P 3 Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible	Not Possible

ANSI Isolation Level SERIALIZABLE

!=

Definition in serializability theory

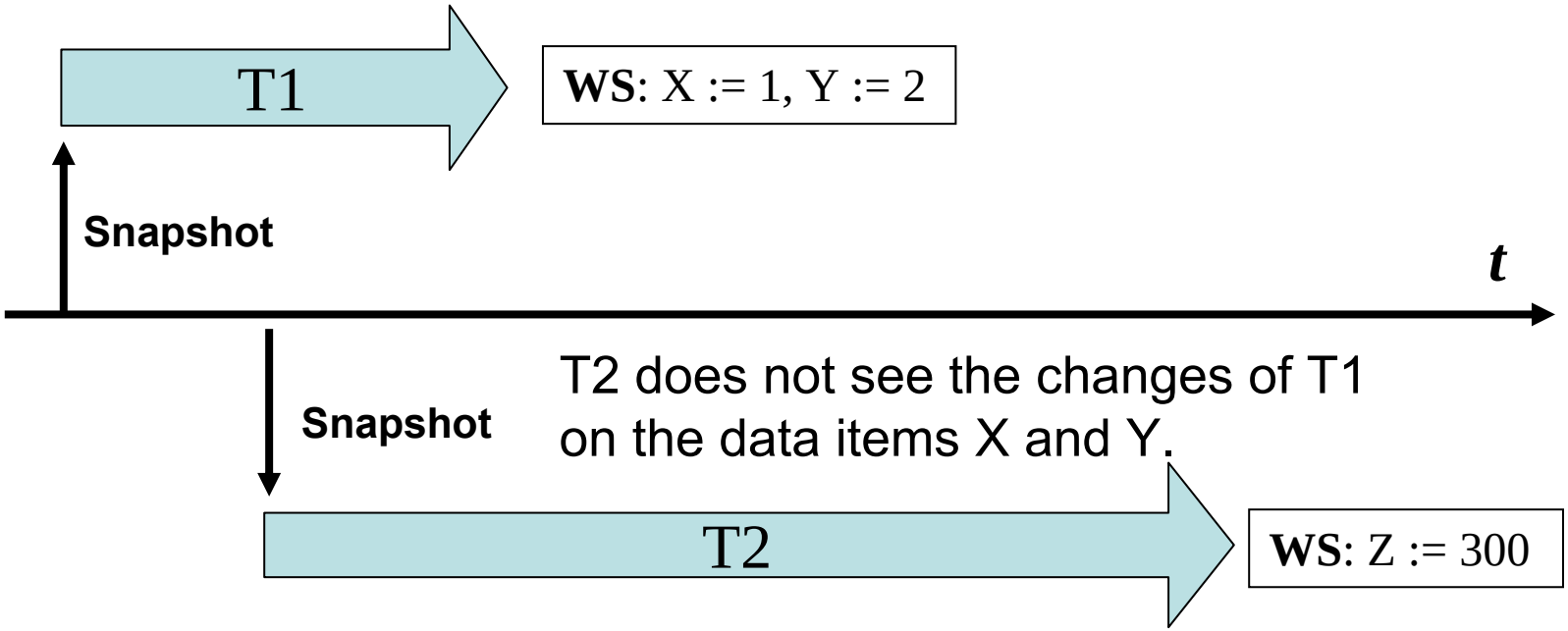
(e.g., conflict serializability)

Snapshot Isolation (SI)

- Multiversion Concurrency Control Mechanism
- Used in PostgreSQL, Oracle and SQL Server 2005
- Readers never conflict with writers ↔ unlike traditional DBMS (e.g., IBM DB2)!
- Does not guarantee „real“ serializability
- But: ANSI “serializability” fulfilled

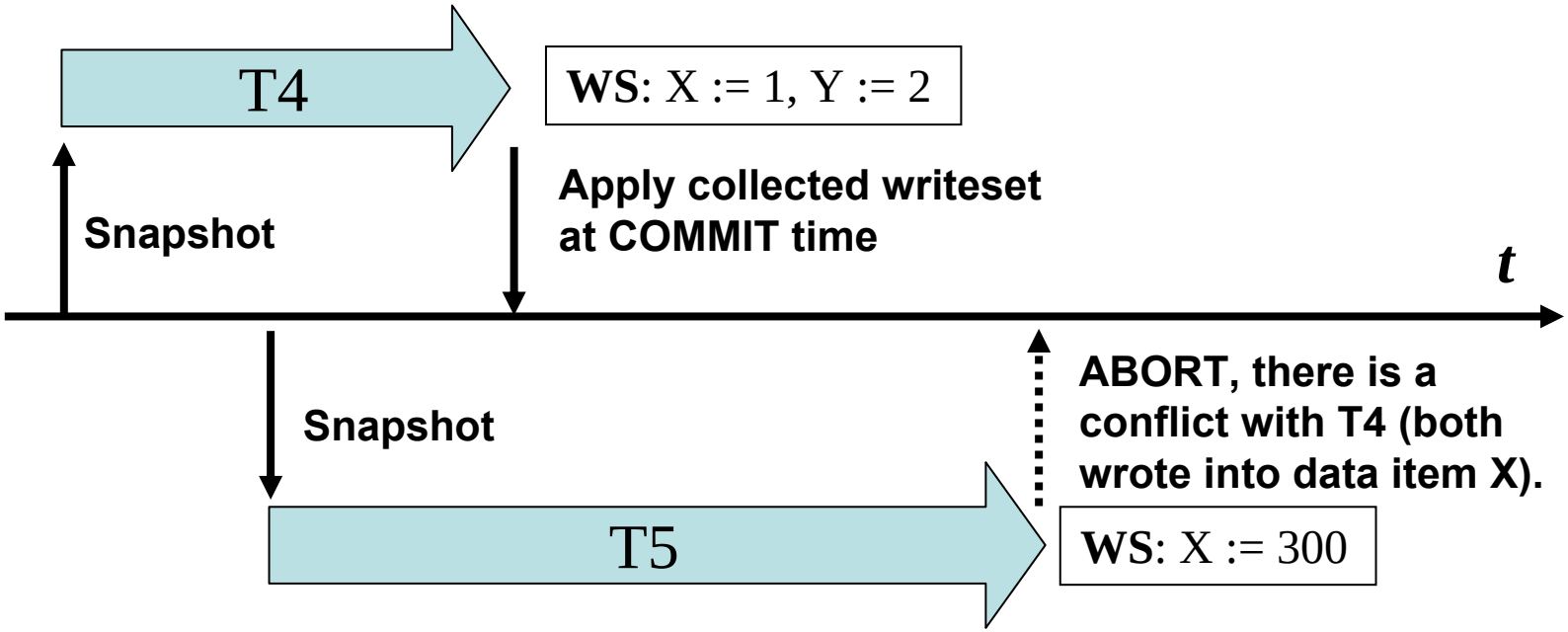
Snapshot Isolation - Basic Idea:

- Every transaction reads from its ***own snapshot*** (copy) of the database (will be created when the transaction starts).
- Writes are collected into a ***writeset (WS)***, not visible to concurrent transactions. Two transactions are considered to be concurrent if one starts (takes a snapshot) while the other is in progress.

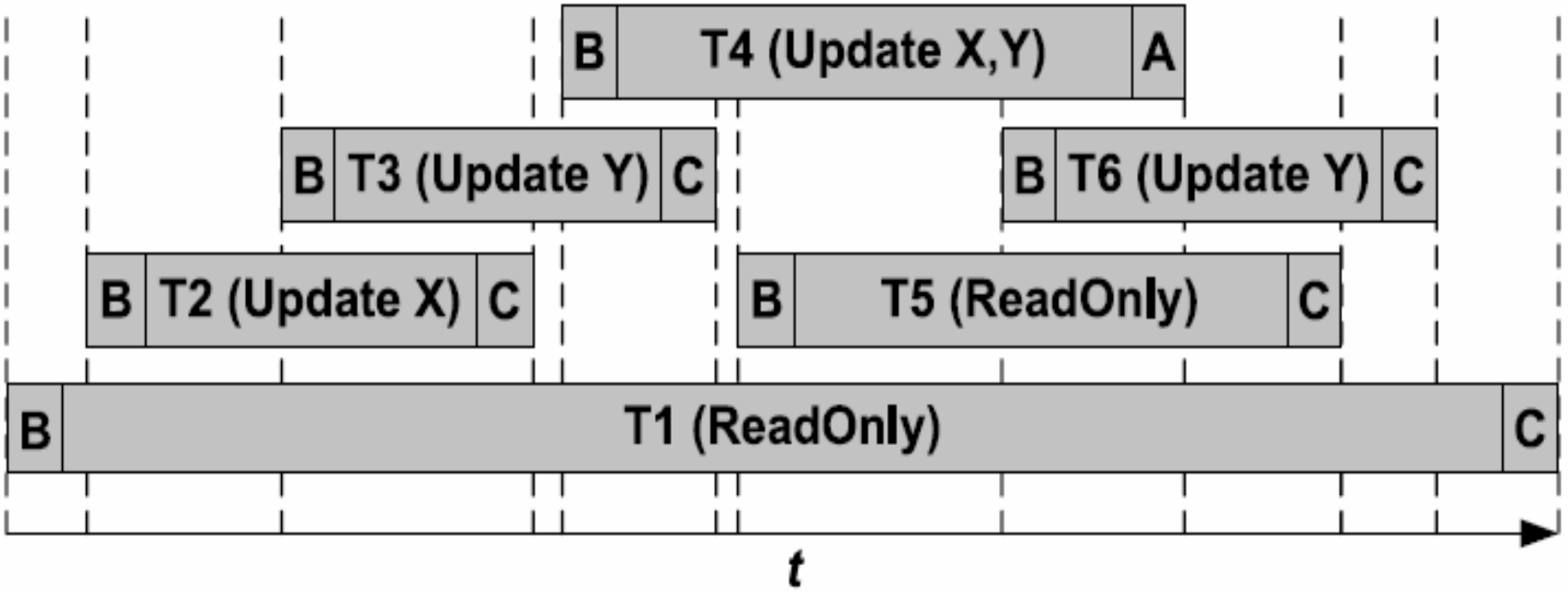


Snapshot Isolation – Conflict Resolution:

- At the commit time of a transaction its writeset WS is ***compared*** to those of concurrent committed transactions. If there is no conflict (overlapping), then the WS can be applied to stable storage and is visible to transactions that begin afterwards.
- However, if there is a conflict with the WS of a concurrent, already committed transaction, then the transaction must be aborted. → **"First Committer Wins Rule"**

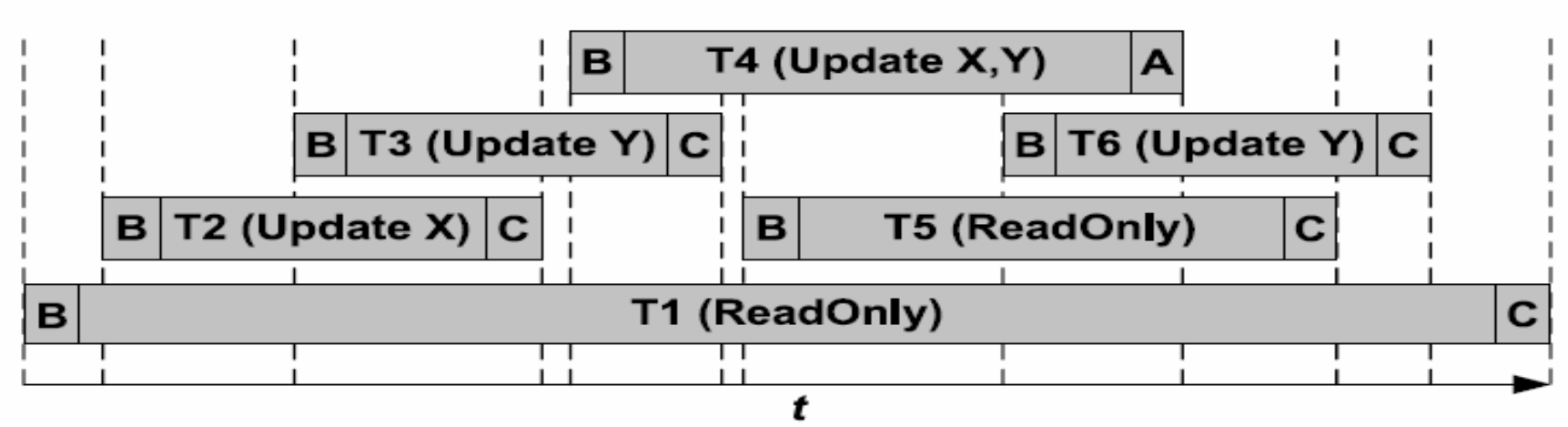


Example Transactions in SI Database



The symbols B, C and A refer to the begin, commit and abort of a transaction

Example Transactions in SI Database (II)



The long running transaction T1 is of type read-only, i.e., its writeset is empty: WS1 = {}. T1 will never conflict with any other other transaction. Updates from concurrent updaters (like T2, T3, T4 and T6) are invisible to T1. T2 will update the database element X, it does not conflict with any other concurrent transaction. T3 updates Y, it does not see the changes made by T2 on X, since it started while T2 was still running. T4 updates X and Y. Conforming to the first-committer-wins rule it cannot commit, since its writeset overlaps with that from T3 and T3 committed while T4 was running. The transaction manager has therefore to abort T4 when the user tries to commit. T5 is read-only and sees the changes made by T2 and T3. T6 can successfully update Y. Due to the fact that T4 did not commit, the overlapping writesets of T6 and T4 do not impose a conflict.

Does SI offer Serializability?

- It avoids the phenomenas P0-P3:
  → ANSI SERIALIZABLE isolation level conform.
- …but ANSI SERIALIZABLE is not the same as defined in traditional serializability theory (conflict serializability).
- **Example:**

  T1: $b_1\ r_1(X)\ w_1(Y)\ c_1$

  T2: $b_2\ r_2(Y)\ w_2(X)\ c_2$

  Schedule: $b_1\ b_2\ r_1(X)\ r_2(Y)\ w_1(Y)\ w_2(X)\ c_1\ c_2$
- **Not conflict equivalent to a serial history, but can happen with Snapshot Isolation.**
- We will come back to that topic later on.

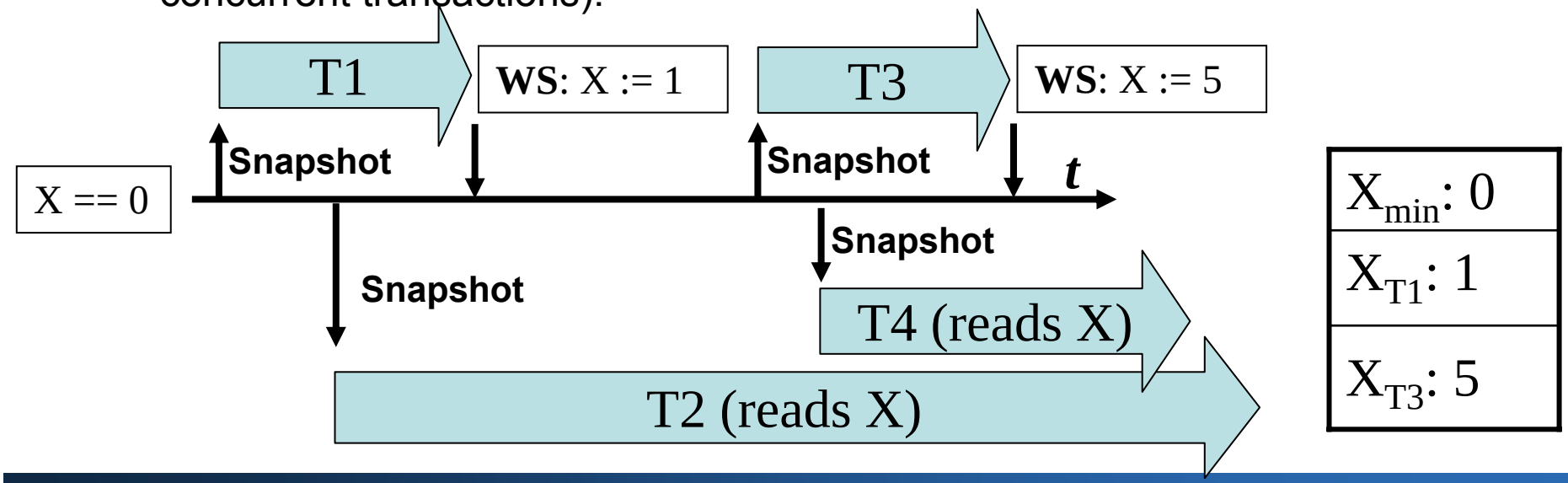
Implementation of SI in real Systems (I)

- Of course, making a copy of the database and managing and comparing (possibly huge) writesets for every transaction is not that efficient…
  E.g., assume a database size of 4GB and a target troughput of 100 concurrent transactions (each having a writeset of 100KB).

- Real SI implementations use an ***incremential*** variant of Snapshot Isolation, using
  - **different versions** of the same data row (to simulate snapshots).
  - **row level (tuple) locks** (to detect write-write conflicts between concurrent transactions).

Implementation of SI in real Systems (II)

- Snapshots are implemented by having ***multiple versions*** (hence, *multi version concurrency control*) of the same data item (e.g., data rows). A transaction that modifies a row generates automatically a new version of this row (which is only visible to transactions that begin (i.e., take a snapshot) after this transaction has committed).
- For every transaction the DB has to decide which version of a data item is „visible“ (e.g, a long running transaction (like T2 below) can see a very old version of a row, even though the row was updated many times by other concurrent transactions).

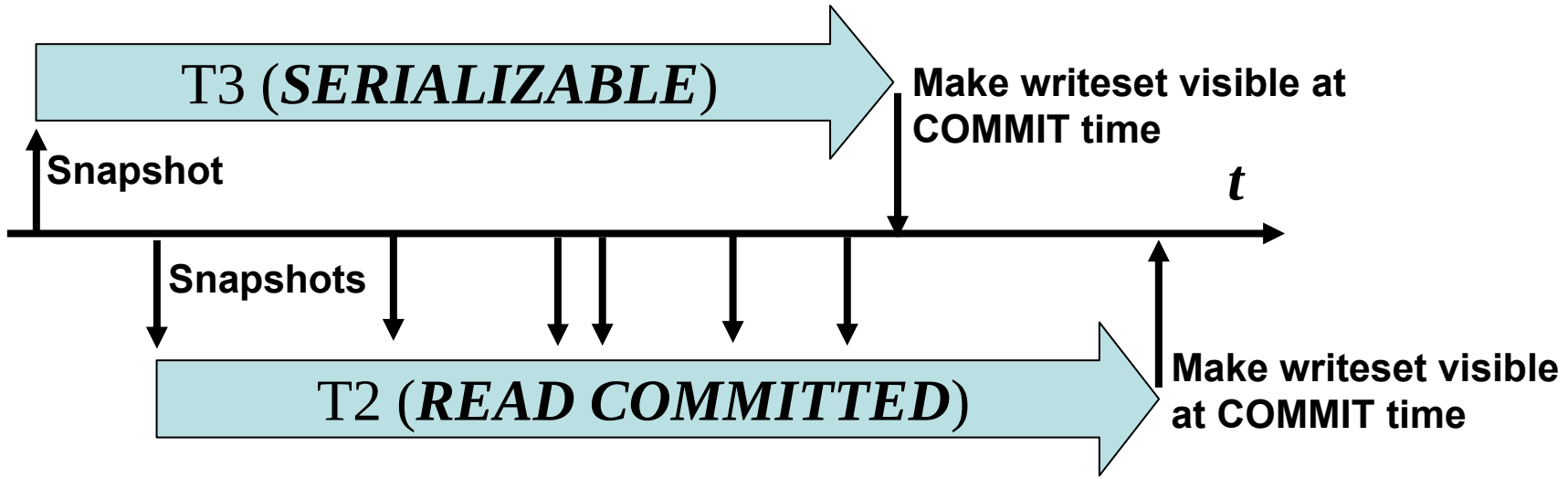


Implementation of SI in real Systems (III)

- Incrementally checking writesets: ***row-level locks***.
- A row-level lock on a row is automatically acquired when the row is updated (or deleted or marked for update). The lock is held until the transaction commits or rolls back. Row-level locks do not affect data querying (of any version of the row); they block writers to the same row only. To acquire a row-level lock on a row without actually modifying the row, the user has to select the row with SELECT FOR UPDATE. Once a particular row-level lock is acquired, the transaction may update the row multiple times without fear of conflicts.
- PostgreSQL does not remember any information about modified rows in memory, so it has no limit to the number of rows locked at one time. However, locking a row may cause a disk write if buffer space is low.

Implementation of SI in real Systems (IV)

- Oracle and PostgreSQL offer two variants of Snapshot Isolation: **SERIALIZABLE** (as described so far) and **READ COMMITTED (**the default isolation level in both products**)**
- READ COMMITTED: the main difference to SERIALIZABLE is the implementation of the snapshot: a transaction running in this isolation mode gets a new snapshot for every issued SQL statement (every statement sees the latest committed values (generated versions) of the database).



SI with Isolation Level SERIALIZABLE

- When a transaction T2 running in isolation level SERIALIZABLE tries to modify a row (i.e., tries to generate a new version of the row) that was modified by a concurrent transaction T1 which has already committed, then the update operation of T2 fails (therefore, not the first committer, but rather the first updater wins).

  → PostgreSQL then also aborts the whole transaction T2.
  → Oracle is more flexible: it allows the user to proceed with other operations in T2.
- If T1 is concurrent but not committed yet (i.e., it holds a row level lock), then both products behave the same: they block transaction T2 until T1 commits or aborts.

  - If T1 commits, then the same things happen as described above.

  - If T1 aborts, then the update operation of T2 can proceed.
- The blocking of a transaction due to a potential update conflict is of course not unproblematic: it can lead to *deadlocks*, which must be resolved by the database (by aborting some of the involved transactions).

SI with Isolation Level READ COMMITTED

- READ COMMITTED is a slightly less strict isolation level. It is the ***default isolation level*** for both Oracle and PostgreSQL.
- A new snapshot is taken for every issued SQL statement (every statement sees the latest committed values).
- The handling of conflicting operations is different than in SERIALIZABLE mode: if a transaction T2 running in READ COMMITTED mode tries to update a row which was already updated by a concurrent transaction T1, then T2 gets blocked until T1 has either committed or aborted.

  - If T1 aborts, no problem, T2 can proceed.

  - If T1 commits, then T2's update statement gets re-evaluated again, since the updated row possibly does not match a used selection predicate anymore.
- READ COMMITTED avoids phenomena P0 and P1, but is vulnerable to P2 and P3 (fuzzy read and phantom).

Comparison Chart for PostgreSQL	READ COMMITTED	SERIALIZABLE
Dirty write	Not Possible	Not Possible
Dirty read	Not Possible	Not Possible
Non-repeatable read	Possible	Not Possible
Phantoms	Possible	Not Possible
Compliant with ANSI/ISO SQL 92	Yes	Yes
Read snapshot time	Statement	Transaction
Row-level locking	Yes	Yes
Readers block writers	No	No
Writers block readers	No	No
Different-row writers block writers	No	No
Same-row writers block writers	Yes	Yes
Waits for blocking transaction	Yes	Yes
Subject to "can't serialize access" error	No	Yes
Error after blocking transaction aborts	No	No
Error after blocking transaction commits	No	Yes

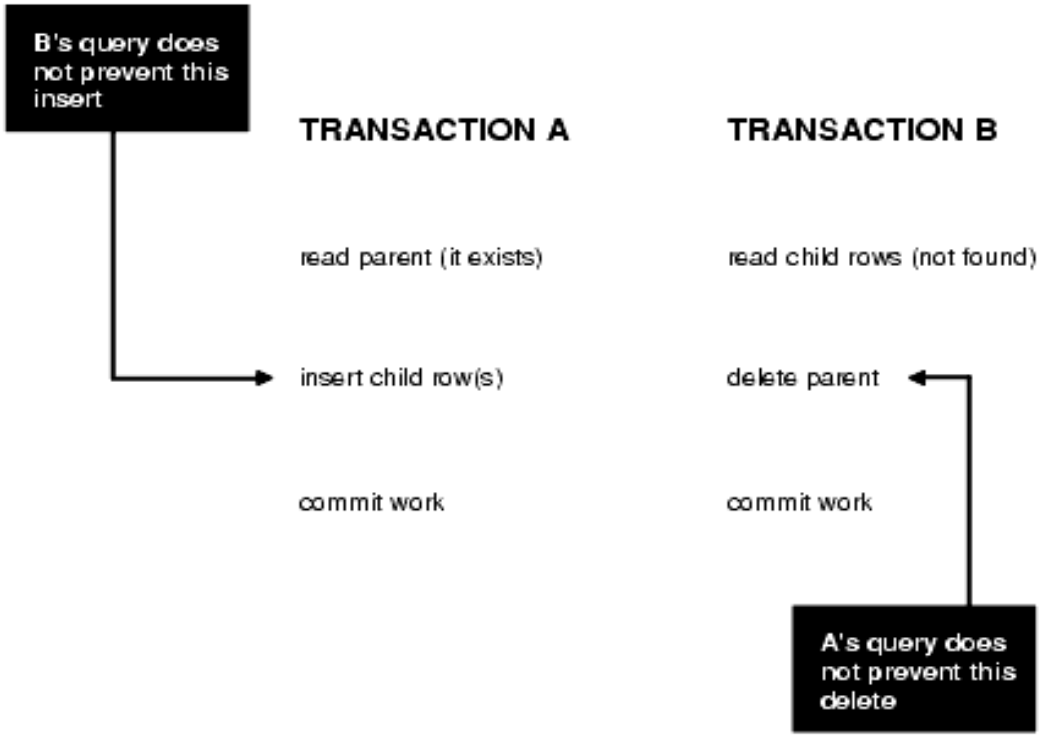
The Remaining Problem: Serializability

- Isolation level SERIALIZABLE avoids the four common phenomenas, but that is not enough: because readers in SI based databases do not lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction.

  In other words, if a row is returned by SELECT it doesn't mean that the row is still current at the instant it is returned (i.e., sometime after the current query began). The row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

- We therefore need sometimes to explicitly lock data to be sure that no other transaction can concurrently modify it. This is very important if we port applications from databases where locks are set for all reads!

Application based Referential Integrity



- A wants to add a child node to a parent (first checks if parent exists)
- B wants to delete a parent (first checks to see if there are no childs)
- Using SI, both can proceed in parallel, the writesets do not overlap.
- **The result is inconsistent: there is a child entry without a parent entry.**

Application based Referential Integrity (II)

- The read issued by transaction A does not prevent transaction B from deleting the parent row, and transaction B's query for child rows does not prevent transaction A from inserting child rows.
- This scenario leaves a child row in the database with no corresponding parent row. This result occurs even if both A and B are SERIALIZABLE transactions, because neither transaction prevents the other from making changes in the data it reads to check consistency.
- As this example shows, sometimes you must take steps to ensure that the data read by one transaction is not concurrently written by another. This requires a greater degree of transaction isolation than defined by SQL92 SERIALIZABLE mode.

General Solutions

- **„Pseudo writes“:** Update values which we do not want to be concurrently modified by reading and immediatelly writing them back. The touched rows now belong to our writeset (we hold row level locks), and so concurrent updates will be detected by the database. Problem: inefficient (imagine we want to make sure that 9000 rows of a table do not get modified, then we have to overwrite 9000 rows with the original content…)
- **SELECT FOR UPDATE**: Better solution, we just mark the rows that we do not want to get concurrently modified (actually we just acquire row level locks), even if we did not change the contents of the rows. The effect is the same: concurrent updates are not possible, they lead to a conflict.
- **Table level Locking**: SELECT FOR UPDATE has some overhead. If we want to lock like 90% of a table, it is perhaps more efficient to lock the whole table. Disadvantages: we also block transactions that are going to work on the other 10%, even though there is absolutely no conflict. **Be careful**: table level locking does not put the whole table in our writeset (no row level locks are acquired), it just blocks other transactions during a certain period of time.

Solving the Referential Integrity Problem

- Transaction A could make use of SELECT FOR UPDATE to query and lock the parent row and thereby prevent transaction B from deleting the row.
- Another possible approach could be to work in READ COMMITTED mode: Transaction B could prevent Transaction A from gaining access to the parent row by reversing the order of its processing steps:
  Transaction B first deletes the parent row, and then rolls back if its subsequent query detects the presence of corresponding rows in the child table. *However, working with guarantees based on the order of statements is not recommended, it makes things very hard to understand, especially if there are many involved transaction patterns.*

And DBMS based Referential Integrity?

 # BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

 # SELECT * from products ;

```
 product_no | name
------------+-------
          1 | Apfel
(1 row)
```

(Our snapshot was generated and we see the „Apfel“ product)

(Now a concurrent transaction deletes the Apfel product and commits)

 # SELECT * from products ;

```
 product_no | name
------------+--------
          1 | Apfel
(1 row)
```

(We still see the „Apfel“, since it is included in our (old) snapshot)

(Now let‘s try and add an order for „Apfel“)

 # INSERT INTO orders VALUES (1, 1);

ERROR: could not serialize access due to concurrent update

CONTEXT: SQL statement "SELECT 1 FROM ONLY "public"."products" x WHERE "product_no" = $1 FOR UPDATE OF x“

Based on example tables from the PostgreSQL documentation:

CREATE TABLE products

(product_no *integer* PRIMARY KEY, name *text*);

CREATE TABLE orders

(order_id *integer* PRIMARY KEY,

product_no *integer* **REFERENCES products (product_no)**);

What happened here? It seems that PostgreSQL was very careful: since there is a referential integrity constraint involved in our „INSERT“ statement, it decided to execute (in the background) a SELECT (combined with FOR UPDATE) to make sure that the corresponding product exists and that nobody else will *concurrently* delete the product. However, that has already happened, and so the SELECT FOR UPDATE fails and therefore PostgreSQL decides to not execute the INSERT statement.

Explicit Table Locking

Instead of using row level locks, we can also lock full tables. Tables can be locked in different modes. Only one transaction at a time can hold a lock on a table using one of the two here described modes:

- **LOCK TABLE EXCLUSIVE (Oracle: SHARE ROW EXCLUSIVE)**

  This mode allows only concurrent readers, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

  Be careful: Oracle allows SELECT FOR UPDATE statements from concurrent transactions, PostgreSQL not!
  → *Porting applications can be difficult.*

- **LOCK TABLE ACCESS EXCLUSIVE (Oracle: EXCLUSIVE)**

  This mode guarantees that the holder is the only transaction accessing the table in any way. This is also the **default lock mode** for LOCK TABLE statements that do not specify a mode explicitly.

- Transactions that write into a table (or do a SELECT FOR UPDATE) automatically acquire a special shared table lock that can be held by many concurrent updaters, however, these shared table level locks will conflict with the the above two kinds locks. See the documentation for details.

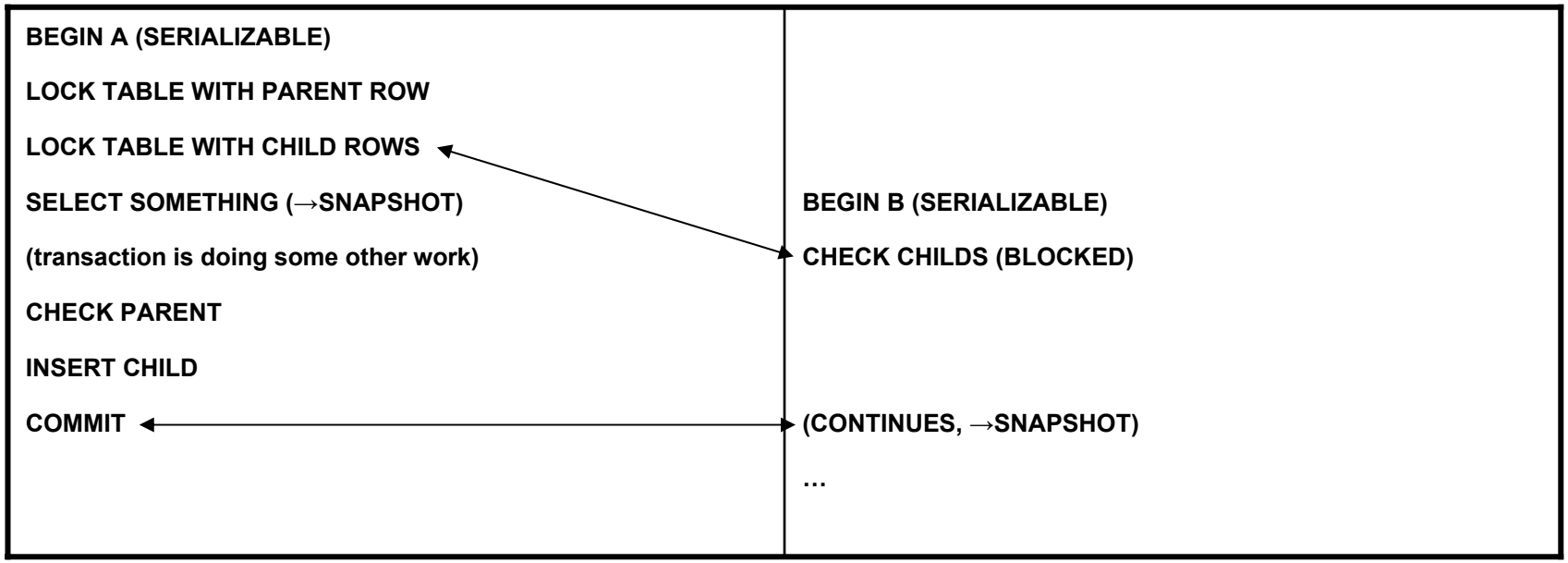
Using Explicit Locking with SERIALIZABLE

- An explicit lock obtained in a serializable transaction guarantees that no other transactions can modify the locked table concurrently, but if the snapshot seen by the transaction predates obtaining the lock, it may predate some now-committed changes in the table. **A serializable transaction's snapshot is actually frozen at the start of its first query or data-modification command** (SELECT, INSERT, UPDATE, or DELETE).

BEGIN A (SERIALIZABLE)	**BEGIN B (SERIALIZABLE)**
SELECT SOMETHING (→SNAPSHOT)	**CHECK CHILDS (→SNAPSHOT)**
(transaction is doing some other work)	**DELETE PARENT ROW**
(transaction is doing some other work)	**COMMIT**
LOCK TABLE WITH PARENT ROW	
LOCK TABLE WITH CHILD ROW	
CHECK PARENT	
INSERT CHILD	
COMMIT	

Using Explicit Locking (II)

- It is possible to obtain explicit locks before the snapshot is frozen.
- Then one is sure that:
  - Nobody modifies the table concurrently
  - We see the latest produced values (not the case on the previous page!)



SELECT FOR UPDATE with NOWAIT (Oracle specific)

Example:

SELECT * FROM items WHERE i_price = 10 FOR UPDATE *NOWAIT*

- With this SELECT we lock all the rows in the items table having a price of 10 bucks. The FOR UPDATE tells Oracle to lock each row as it processes it. Note that this statement does not actually update any data. The row level locks are removed when we commit or roll back.
- The NOWAIT keyword specifies that we do not want the statement to wait until a concurrent transaction already holding a conflicting lock either commits or aborts.
- This means that if the statement detects that the table or any row in the result set is locked it will return an error code and not wait for the lock to be removed.

Want to try it out?

- Log in concurrently (with two sessions) on your PostgreSQL DB.
- To get a feeling for READ COMMITTED and SERIALIZABLE, do the following:
- Try to execute two concurrent transactions that work on the same *data row*, so that one of them gets blocked until the first one has committed.
- Try to achieve (a) that the blocked transaction can continue, and (b), in a second try, that it is aborted when the first transaction commits.
- Also, try put the two transactions in a DEADLOCK condition.

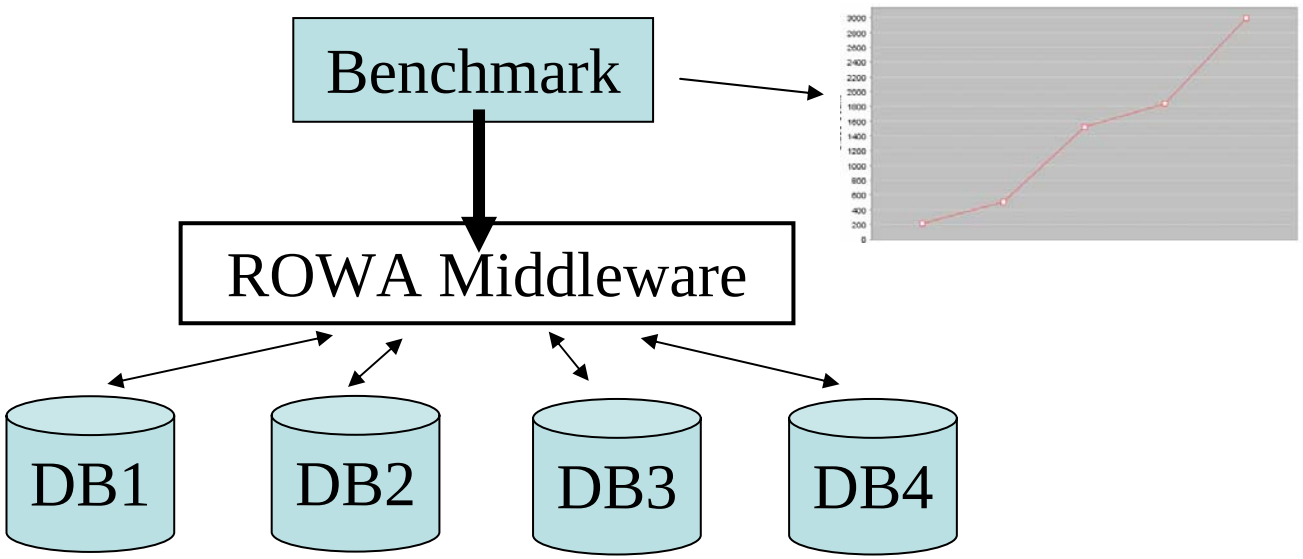
ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction to the Mandatory Exercise III

Task Description

- Your task: Implementation of a Distributed Database.
  - In other words: implement a ROWA based middleware in Java that handles transactions over a set of fully replicated PostgreSQL databases. Construct a benchmark application that shows the performance of your implementation.



Middleware Details

- The Middleware offers a simple interface that can be used by clients (e.g., the benchmark) to execute transactions.
- Transactions are executed by the Middleware over a set of JDBC connections to the databases.
- Read operations have to be performed on one database, updates have to be sent to all databases.
- Since we use PostgreSQL databases, you can use the 2PC features of PostgreSQL to easily implement distributed commits.

Benchmark Details

- We use a 2-factorial benchmark:
  - we use different loads (they differ in the read/write fraction)
  - we use different numbers of databases
- Output of the benchmark can, e.g., be presented with the jfreechart library (use either Swing, SWT or a HTML page to present the results).
- For each experiment, the benchmark software initializes a set of middleware objects (also confguring each with the number of databases to use) and then sends transactions in parallel to the middleware (and measures response times). Use a maximum (!) of 10 worker threads that send transactions to the middleware objects.

Setup

- Each student has 30 databases (each on a separate machine). Please ***DO NOT*** open more than 10 PostgreSQL connections concurrently to the same machine (each machine has a limit of about 1000 concurrent connections, please be fair).
- These PostgreSQL databases can be accessed from all networks at ETH (otherwise use VPN).
- Since you work in groups, it does not matter which of the databases available to your group you use.
- No, it is not necessary to assign special accounts and passwords (see material on the website).

Administrative Things

- The exercise should be solved in groups (2-4 people), divide the task (benchmark (load generation, graph generation), middleware, database setup).
- Presentation in two weeks (in the IFW computer rooms during the lecture on friday).
- Details (Eclipse template project etc.) will be posted soon on http://www.inf.ethz.ch/personal/plattner/vs