Mobisnap: a database system for mobile computing

Nuno Preguiça
DI – FCT
Universidade Nova de Lisboa

Project partially supported by Praxis XXI/FCT/MCT
LSDCS, CTTI, UNL + GSD, UMinho

Motivation
- Widespread use of mobile computers/devices and wireless communications
- Periods of disconnection (from server)
  - No connectivity (no network or network failure)
  - Limited energy (batteries)
  - Economic factors
  - Unacceptable latency (physics)
  - Service overload (during demand peaks)
- Many applications use relational databases

Goal
- System to support independent operation
  - Users should be able to read/write data independently (i.e. without coordination)
- Data structured according to the relational data model
- Based on standards – SQL and PL/SQL

Basic approach
- Optimistic replication
  - Clients partially replicate data
  - Users may submit transactions in disconnected clients

Optimistic replication
Database initial state in server

Client 1

Client 2

Optimistic replication
Client 1 caches objects

Client 1

Server

Client 2
Optimistic replication
Client 1 uploads changes

Operation can be replayed without problems because the object has the same value as seen in the client.

Optimistic replication
Client 2 uploads changes

Conflict !!!
The object has been modified concurrently.

Optimistic replication
Client 2 uploads changes

Problem 1
Detect conflicts

Problem 2
Solve conflicts
Outline
- Introduction
- Optimistic replication and problems
- Middleware architecture
- Mobile transactions
- Reservations
- Reconciliation
- Final remarks

Single server
Maintains official data version

Multiple clients
Maintain database snapshots
- Two versions: committed and tentative
Can read/write the database
Mobile transactions

- Small PL/SQL programs
  - Subset of PL/SQL with some extensions/changes
  - Major restriction: no loops, no cursors
- Programs are tentatively executed in the mobile devices
- Programs are executed (again) in the server to obtain the final result
- Transactions are not integrated using the read/write sets of client execution

DECLARE
l_stock INTEGER;
l_price FLOAT;
BEGIN
SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
IF l_price <= 100 AND l_stock >= 10 THEN
  UPDATE products SET stock = stock - 1 WHERE id = 80;
ENDIF;
SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 47;
IF l_price <= 110 AND l_stock >= 10 THEN
  -- update stock, add order record, send notification
  COMMIT (47,l_price);
ENDIF;
NOTIFY('SMS', '351927435456','Order failed...');
ROLLBACK;
END;

DECLARE
l_stock INTEGER;
l_price FLOAT;
BEGIN
SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
IF l_price <= 100 AND l_stock >= 10 THEN
  UPDATE products SET stock = stock - 1 WHERE id = 80;
ENDIF;
SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 47;
IF l_price <= 110 AND l_stock >= 10 THEN
  -- update stock, add order record, send notification
  COMMIT (47,l_price);
ENDIF;
NOTIFY('SMS', '351927435456','Order added for product red...');
END;
Reservations

- Goal: guarantee the final result of mobile transactions independently
- A reservation provides some guarantee about the future state of the database

Escrow reservation

- Exclusive right to use a share of a partitionable resource represented by a numerical item
  - E.g. the stock of some product may be split among several users
- Implements escrow model in SQL with some changes
  - $x \times \min \Rightarrow \chi$; $x \times \min \land x = \sum(x \times \min)$
  - Traditional: $x \times \min \Rightarrow \chi$; $\chi \geq \min \land x = \sum \chi \land x = \sum \min$.

Value-use reservation

- Right to use some value for a given data item (despite its value when the transaction is executed)
  - E.g. a salesperson may use a reserved price for some product
- Select statements will return the reserved value

Lock-like reservations

- Value-change
  - Exclusive right to modify some data item
    - Traditional fine-grain write lock
    - E.g. a user may reserve the right to change the passenger’s name for a given seat in a train
- Slot
  - Exclusive right to modify records that conform to some given condition
    - Similar to predicate lock (locks records that exist and that do not exist)
    - E.g. a user may reserve the right to schedule a meeting in a room in a given period of time

Shared lock-like reservations

- Shared value-change reservation
  - Guarantees that it is possible to modify some data item
    - E.g. used to guarantee that a counter may be incremented
- Shared slot reservation
  - Guarantees that it is possible to modify data items with some given values
    - E.g. used to guarantee insert operations in an append-only table
**Reservation model**

- Clients obtain (leased) reservation before disconnecting
  - The server enforces the granted reservations updating the database and setting appropriate triggers
- System transparently verifies if transactions can be guaranteed by the available reservations
  - no special operations are used to access reserved data items
  - all reservations are used
- If a transaction is guaranteed in the client, its execution in the server will follow the expected path (if reservations do not expire)

**Example: product request**

### Client gets reservations

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>15</td>
<td>&gt;=0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value-use</td>
<td>products</td>
<td>price</td>
<td>id=80</td>
<td>96.5</td>
<td>---</td>
<td>45-1</td>
</tr>
</tbody>
</table>

BEGIN

SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
IF l_price <= 100 AND l_stock > 10 THEN
UPDATE products SET stock = stock - 10 WHERE id = 80;
INSERT INTO orders VALUES (newid,l785,80,10,l_price,'processing');
COMMIT (80,l_price);
END;
ROLLBACK;
END;

### select statement

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>15</td>
<td>&gt;=0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value-use</td>
<td>products</td>
<td>price</td>
<td>id=80</td>
<td>96.5</td>
<td>---</td>
<td>45-1</td>
</tr>
</tbody>
</table>

BEGIN

SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
IF l_price <= 100 AND l_stock > 10 THEN
UPDATE products SET stock = stock - 10 WHERE id = 80;
INSERT INTO orders VALUES (newid,l785,80,10,l_price,'processing');
COMMIT (80,l_price);
END;
ROLLBACK;
END;

### IF instruction

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>15</td>
<td>&gt;=0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value-use</td>
<td>products</td>
<td>price</td>
<td>id=80</td>
<td>96.5</td>
<td>---</td>
<td>45-1</td>
</tr>
</tbody>
</table>

BEGIN

SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
IF l_price <= 100 AND l_stock > 10 THEN
UPDATE products SET stock = stock - 10 WHERE id = 80;
INSERT INTO orders VALUES (newid,l785,80,10,l_price,'processing');
COMMIT (80,l_price);
END;
ROLLBACK;
END;

### update statement

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>15</td>
<td>&gt;=0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value-use</td>
<td>products</td>
<td>price</td>
<td>id=80</td>
<td>96.5</td>
<td>---</td>
<td>45-1</td>
</tr>
</tbody>
</table>

BEGIN

SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
IF l_price <= 100 AND l_stock > 10 THEN
UPDATE products SET stock = stock - 10 WHERE id = 80;
INSERT INTO orders VALUES (newid,l785,80,10,l_price,'processing');
COMMIT (80,l_price);
END;
ROLLBACK;
END;

l_stock=15; reserved; 45-1
l_price=96.5; reserved; 45-2
Example: product request
update statement

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>5</td>
<td>&gt;0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BEGIN
SELECT stock
IF l_price < 80;
UPDATE products SET stock = stock - 10 WHERE id = 80;
INSERT INTO orders VALUES (newid,8785,80,10,l_price,‘processing’);
COMMIT (80,l_price);
END;
ROLLBACK;
END;

Update can be guaranteed by the escrow reservation
Updates reservation and database snapshots

l_stock=15; reserved; 45-1
l_price=96.5; reserved; 45-2

Example: product request
insert statement

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>5</td>
<td>&gt;0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BEGIN
INSERT INTO values
WHERE id = 80;
Execution proceeds
INSERT INTO orders VALUES (newid,8785,80,10,l_price,‘processing’);
COMMIT (80,l_price);
END;
ROLLBACK;
END;

l_stock=15; reserved; 45-1
l_price=96.5; reserved; 45-2

Example: product request
commit statement

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>table</th>
<th>column</th>
<th>condition</th>
<th>value</th>
<th>info</th>
<th>linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>45-1</td>
<td>escrow</td>
<td>products</td>
<td>stock</td>
<td>id=80</td>
<td>5</td>
<td>&gt;0</td>
<td>---</td>
</tr>
<tr>
<td>45-2</td>
<td>value=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BEGIN
transactions ends
RESULT: reservation commit; level=read
Meaning: the execution in the server will follow the same execution path but write statement may be blocked (the insert statement was not guaranteed)
COMMIT (80,l_price);
END;
ROLLBACK;
END;

l_stock=15; reserved; 45-1
l_price=96.5; reserved; 45-2

Transaction execution in the client
1st — checks if it can be guaranteed

- If the execution ends in a commit statement the result is reservation commit, level:
  - Full
    - All statements could be guaranteed
  - Read
  - Pre-condition
    - It was possible to guaranteed the result of IF statements in the depth-first execution path
  - Alternative pre-condition
    - It was possible to guarantee an alternative execution path (some IF statement could not be guaranteed)

Transaction execution in the client
2nd — when it can not be guaranteed

- If the execution requires some uncached data item, the result is unknown
- If the first execution ends in a rollback statement, the transaction is re-executed against the tentative data version
  - If execution ends in a commit statement, the result is tentative commit
  - If execution ends in a rollback statement, the result is tentative commit

Transaction execution in the server

- Re-executes the transactions
- For reserved transactions
  - Unlocks data items of used reservations
  - Guaranteed reads need special handling (the same value must be returned as the result of a select statement)
  - "Select" may read multiple columns – the same must be returned
Remarks
- Reservations may lead to unnecessary aborts
  - Re-execution mechanism + reconciliation
- Clients have to obtain the right reservations before disconnecting
  - Application-dependent

Mobile sales application
- Good prediction
  - Local guarantee: > 90% maximum
    - Tend to 100% as usage rate deviates from 100%
  - Immediate guarantee: > 95% maximum
- No prediction
  - Local guarantee: > 80% maximum
  - Immediate guarantee: > 95% maximum

Contributions
- Multiple types of reservations
  - Necessary to guarantee most transactions
  - Implemented in a SQL-based system
- Client transparently verifies if reservations can be used
  - Transactions written as usual
  - Integrated with mobile transactions
- Middleware approach
  - Legacy clients can continue to access database as usual

Outline
- Introduction
- Optimistic replication and problems
- Middleware architecture
- Mobile transactions
- Reservations
- Reconciliation
- Final remarks

Reconciliation
- The execution order may influence the
  the result and the number of transactions that can be executed successfully
  - Transactions waiting re-execution
  - Client may upload transactions from other clients
Log relations (constraints)
- Applications may explicitly set the following relations among transactions
  - Predecessor-successor
  - Alternatives
    - May be extracted from single transaction code
  - Parcel
    - Original IceCube

Object relations (constraints)
- From the transaction code, the following relations are inferred
  - Overlaps
  - Commutes
  - Makes impossible
    - T1 write make T2 preconditions false
  - Helps
    - T1 writes helps T2 preconditions to be true
  - Prejudices

Reconciliation approach
- Based on IceCube (MSR, Cambridge)
- Goal: maximize the set of transactions executed with success
- Reconciliation process seen as an optimization problem
  - Set of (semantic) static relations defined among transactions
  - Heuristic search explores static information

Reconciliation example
Log 1
- 9am room A or 9am room B
- 9am room C

Log 2
- 9am room A or 9am room B
- 9am room C

Possible schedules
- 9am room A or 9am room B
- 9am room C
Reconciliation algorithm

- New schedule created iteratively. In each step:
  - Selects a transaction (using an heuristic based on the static relations)
  - Executes the transaction
- Reconciliation creates new schedules until a good solution is found

How to infer relations?

1st – extract information

- Analyze each program statically, extracting from paths leading to commits:
  - (semantic) read set
    - `select` statements used in writes/preconditions
  - (semantic) write set
    - `Update`, `insert`, `delete` statements
  - (semantic) preconditions
    - Conditions in `if` statements leading to a commit
SELECT stock, price INTO l_stock, l_price FROM products WHERE id = 80;
-- l_stock = read(products, id=80, stock)
-- l_price = read(products, id=80, price)
IF l_price <= 100 AND l_stock >= 10 THEN
  -- as the contrary leads to a rollback
  -- precond(read(products, id=80, stock) >= 10)
  -- precond(read(products, id=80, price) < 100)
  UPDATE products SET stock = l_stock - 10 WHERE id = 80;
  -- update(products, id=80, stock, stock-10)
INSERT INTO orders VALUES (newid, 8785, 80, 10, l_price, 'processing');
-- insert(orders, (id, client, product, qty, price, status),
-- (newid, 80, 10, l_price, 'processing'))
COMMIT;
ELSE
ROLLBACK;
ENDIF;
ReadSet = (read(products, id=80, stock), read(products, id=80, price))

Example: overlaps

- T1 overlaps with T2 if any of the following is true:
  - T1 reads a data item written (insert, update, delete) by T2 (or vice-versa)
  - T1 writes a data item written by T2 (or vice-versa)
    - Unless blind deletes

- More generally, we should also consider reads/writes that access different data items related by some constraint

Remarks

- IceCube may improve the reconciliation result
- The automatic extraction of relations allows the use of the IceCube approach with generic transactions

Final remarks

- Mobile transactions
  - Define conflict detection and resolution rules
- Reservations
  - Used to guarantee the result of transactions in mobile (disconnected) clients
- Reconciliation
  - Optimizes the set of transactions that can be executed with success

Questions?

http://asc.di.fc.ul.pt/mobisnap
nmp@di.fc.ul.pt