CS848 - Cloud Data Management

Introduction and Background

Ken Salem

David R. Cheriton School of Computer Science
University of Waterloo

Winter 2010

What is cloud computing?

- It seems that everybody who is offering an internet service or using a cluster wants to label themselves “cloud”

What is cloud computing?

- It seems that everybody who is offering an internet service or using a cluster wants to label themselves “cloud”
- Adjectives associated with clouds
  - scalable
  - highly-available
  - pay-as-you-go
  - on demand

What is cloud computing?

- It seems that everybody who is offering an internet service or using a cluster wants to label themselves “cloud”
- Adjectives associated with clouds
  - scalable
  - highly-available
  - pay-as-you-go
  - on demand
- Not much point in trying to pin down what is cloud and what is not.

Services Spectrum

less flexible
more constrained
less effort

more flexible
less constrained
more effort

Google docs
TurboTax
salesforce.com
Windows Azure
amazon web services
Eucalyptus
the rackspace cloud

Services Spectrum

less flexible
more constrained
less effort

more flexible
less constrained
more effort

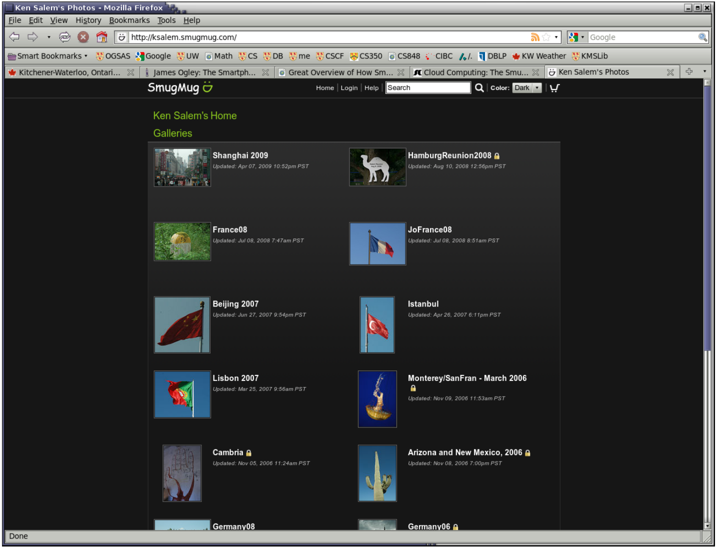
⟵ ⟶

Google docs
TurboTax
salesforce.com
Windows Azure
amazon web services
Eucalyptus
the rackspace cloud

software–as–a–service

servers–as–a–service

A Cloud User


Ken Salem's Photos - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://ksalem.smugmug.com/
Smart Bookmarks OGSAS Google UW Math CS DB me CSCF CS350 CS848 CIBC /. DBLP KW Weather KMSLib
Kitchener-Waterloo, Ontari...
James Ogley: The Smartph...
Great Overview of How Sm...
Cloud Computing: The Smu...
Ken Salem's Photos
SmugMug
Home Login Help Search Color: Dark
Ken Salem's Home
Galleries
Shanghai 2009
Updated: Apr 07, 2009 10:52pm PST
HamburgReunion2008
Updated: Aug 10, 2008 12:56pm PST
France08
Updated: Jul 08, 2008 7:47am PST
JoFrance08
Updated: Jul 08, 2008 8:51am PST
Beijing 2007
Updated: Jun 27, 2007 9:54pm PST
Istanbul
Updated: Apr 26, 2007 6:11pm PST
Lisbon 2007
Updated: Mar 25, 2007 9:56am PST
Monterey/SanFran - March 2006
Updated: Nov 09, 2006 11:53am PST
Cambria
Updated: Nov 05, 2006 11:24am PST
Arizona and New Mexico, 2006
Updated: Nov 08, 2006 7:00pm PST
Germany08
Germany06
Done

External Cloud Services

- Benefits
  - pay-as-you-go eliminates capital costs
  - economies of scale lower operating costs (hardware procurement, networking, power, administration)
  - arbitrary scalability ($\$100 = 1$ server for 1000 hours $=$ 1000 servers for 1 hour)
    - bursty service loads
    - massively-parallel analytics
- Drawbacks
  - communication latency and bandwidth
  - autonomy and trust
  - data security and privacy

In-House Clouds

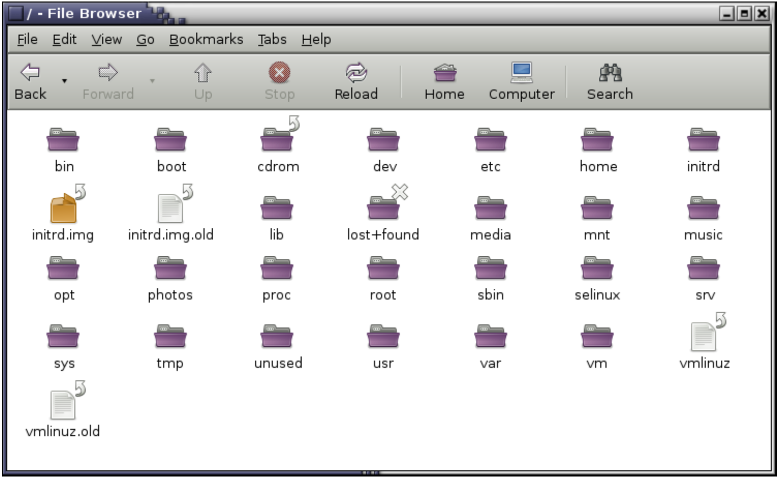
- consolidate physical resources
  - higher utilizations, lower costs
- instant and flexible provisioning for new projects and services
- compatibility with external public clouds

EC2/Eucalyptus Basics

- images and instances
- management
- storing data

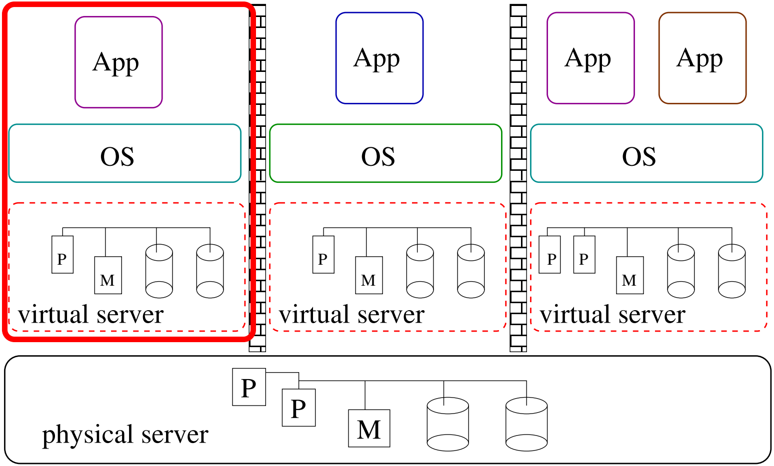
Images

An image is a signed, encrypted snapshot of a root file system.



Instance

An instance is a virtual machine.



Instance Types

Instances come in different types.

Type	VCPU	ECU	GB	I/O	$/hr
S	1	1	1.7	Mod	0.085
L	2	4	7.5	High	0.340
XL	4	8	15	High	0.680
HighC XL	8	20	7	High	0.680
HighM XXXXL	8	26	68.4	High	2.400

Pricing for Linux Amazon EC2 instances in N.Va. region as of Dec 4 2009.

Performance Guarantees in the Cloud

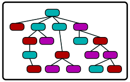
Amazon on instance performance

One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. . . . To find out which instance will work best for your application, the best thing to do is to launch an instance and benchmark your own application.

Amazon on I/O performance

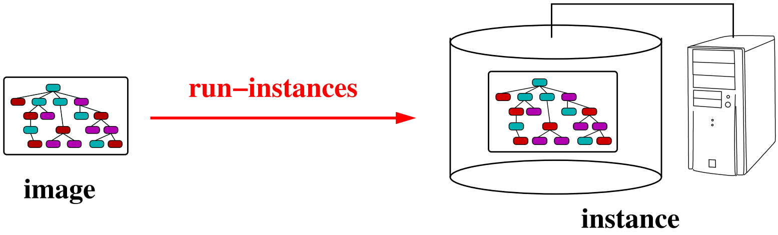
Each of the instance types has an I/O performance indicator (moderate or high). Instance types with high I/O performance have a larger allocation of shared resources.

Instance Management

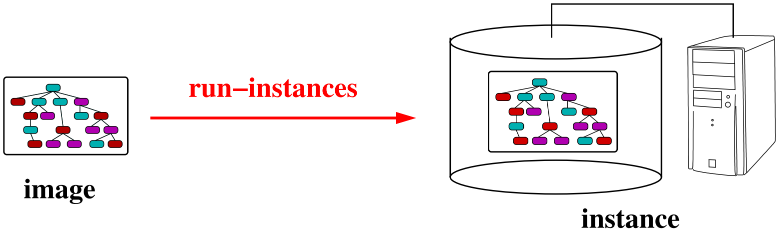


image

Instance Management



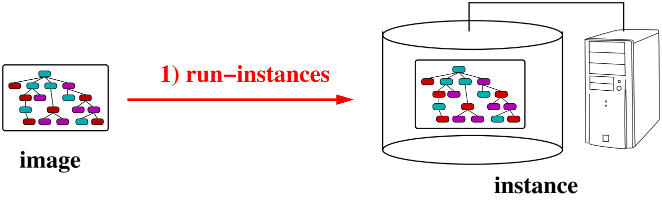
Instance Management



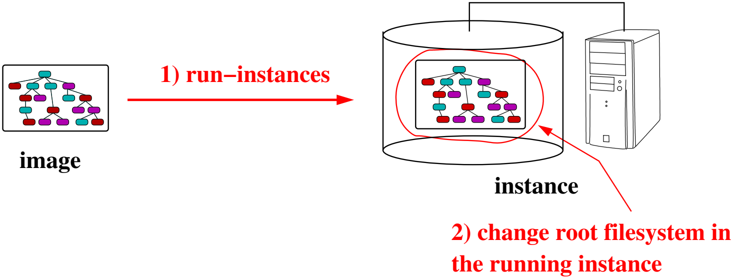
Once an instance is running:

- manage it (reboot, terminate, monitor . . .)
- attach persistent storage to the the instance
- manage network access to the instance
- log in!

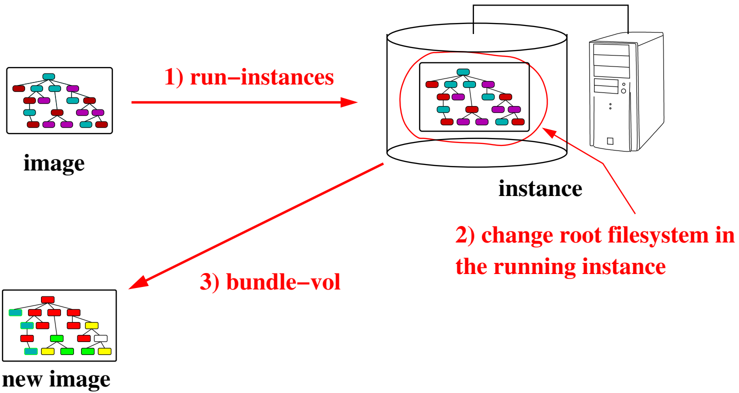
Authoring Images



Authoring Images



Authoring Images



Value-Added Services

- storage services
- management dashboards (e.g., RightScale)
- monitoring
- automated provisioning and load balancing
- specialized instances, e.g., Amazon Relational Database Service

Storing Data

- instance storage (ephemeral)

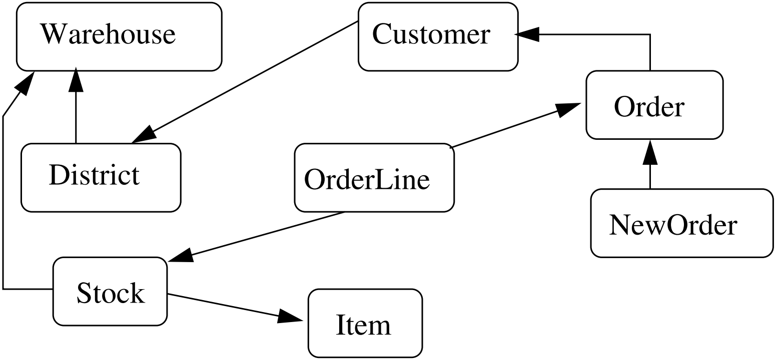
Storing Data

- instance storage (ephemeral)
- Elastic Block Storage (EBS)
    - named, persistent, reliable volumes
    - block level access (looks like a disk)
    - can be attached to a running instance

Storing Data

- instance storage (ephemeral)
- Elastic Block Storage (EBS)
    - named, persistent, reliable volumes
    - block level access (looks like a disk)
    - can be attached to a running instance
- network storage services
    - S3/Walrus
    - SimpleDB, BigTable, PNUTS (and more . . .)

The TPC-C Database



The TPC-C NewOrder Operation

- A NewOrder operation places an order for one or more items for a given customer from a given warehouse.
- steps:
  - read tax and discount rates from warehouse, district and customer tables
  - insert new 1 new tuple in each of the order and neworder tables
  - for each item:
    - read the price from the item table
    - read and update the stock level in the stock table
    - insert a tuple into the orderline table
- executing NewOrder as a transaction ensures that it is atomic

The TPC-C Payment Operation

- A Payment operation records a payment on a customer's account
- steps:
  - update customer total payments and payment count fields in the customer table
  - update total payments field in district table
  - update total payments field in warehouse table

Transaction Properties

- Transactions are *durable*, *atomic* application-specified units of work.

  Atomic: indivisible, all-or-nothing.
  Durable: effects survive failures.

“ACID” Properties of Transactions

A tomic: a transaction occurs entirely, or not at all

C onsistent

I solated: a transaction’s unfinished changes are not visible to others

D urable: once it is complete, a transaction’s changes are permanent

Abort and Commit

A transaction may terminate in one of two ways:

commit: When a transaction *commits*, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.

abort: When a transaction *aborts*, any updates it may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.

Serializability

- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order. If T_i and T_j are concurrent transactions, then either:
  - T_i will appear to precede T_j, meaning that T_j will "see" any updates made by T_i, and T_i will not see any updates made by T_j, or
  - T_i will appear to follow T_j, meaning that T_i will see T_j's updates and T_j will not see T_i's.

Serializability: An Example

- An serial execution of two transactions, T_1 and T_2:

$$H_b = w_1[x]\ w_1[y]\ r_2[x]\ r_2[y]$$

Serializability: An Example

- An serial execution of two transactions, T_1 and T_2:

$$H_b = w_1[x]\, w_1[y]\, r_2[x]\, r_2[y]$$

- An equivalent interleaved execution of T_1 and T_2:

$$H_a = w_1[x]\, r_2[x]\, w_1[y]\, r_2[y]$$

Serializability: An Example

- An serial execution of two transactions, T_1 and T_2:

$$H_b = \; w_1[x]\, w_1[y]\, r_2[x]\, r_2[y]$$

- An equivalent interleaved execution of T_1 and T_2:

$$H_a = w_1[x]\, r_2[x]\, w_1[y]\, r_2[y]$$

- An interleaved execution of T_1 and T_2 with no equivalent serial execution:

$$H_c = \; w_1[x]\, r_2[x]\, r_2[y]\, w_1[y]$$

Serializability: An Example

- An serial execution of two transactions, T_1 and T_2:

$$H_b = \; w_1[x]\; w_1[y]\; r_2[x]\; r_2[y]$$

- An equivalent interleaved execution of T_1 and T_2:

$$H_a = w_1[x]\; r_2[x]\; w_1[y]\; r_2[y]$$

- An interleaved execution of T_1 and T_2 with no equivalent serial execution:

$$H_c = \; w_1[x]\; r_2[x]\; r_2[y]\; w_1[y]$$

H_b is serializable because it is equivalent to H_a , a serial schedule. H_c is not serializable.

Two-Phase Locking

- The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     - a *shared lock* is required to read an object
     - an *exclusive lock* is required to write an object

Two-Phase Locking

- The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     - a *shared lock* is required to read an object
     - an *exclusive lock* is required to write an object
  2. Two or more transactions may not hold locks on the same object unless all hold shared locks.

Two-Phase Locking

- The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     - a *shared lock* is required to read an object
     - an *exclusive lock* is required to write an object
  2. Two or more transactions may not hold locks on the same object unless all hold shared locks.
  3. Once a transaction has released (unlocked) any object, it may not obtain any new locks. (In strict two-phase locking, locks are held until the transaction commits or aborts.)

Two-Phase Locking

- The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     - a *shared lock* is required to read an object
     - an *exclusive lock* is required to write an object
  2. Two or more transactions may not hold locks on the same object unless all hold shared locks.
  3. Once a transaction has released (unlocked) any object, it may not obtain any new locks. (In strict two-phase locking, locks are held until the transaction commits or aborts.)

Theorem

If all transactions use two-phase locking, the resulting execution history will be serializable.

Snapshot Isolation (SI)

- each transaction T has a start time ($start(T)$) and a commit time ($commit(T)$) - unless it aborts.

Snapshot Isolation (SI)

- each transaction T has a start time ($start(T)$) and a commit time ($commit(T)$) - unless it aborts.
- each transacation T “sees” a snapshot of the database that includes all updates of transactions that commit before $start(T)$ and no updates of transactions that commit after $start(T)$, except . . .

Snapshot Isolation (SI)

- each transaction T has a start time ($start(T)$) and a commit time ($commit(T)$) - unless it aborts.
- each transacation T “sees” a snapshot of the database that includes all updates of transactions that commit before $start(T)$ and no updates of transactions that commit after $start(T)$, except . . .
- . . . that T sees its own updates.

Snapshot Isolation (SI)

- each transaction T has a start time (*start*(T)) and a commit time (*commit*(T)) - unless it aborts.
- each transacation T "sees" a snapshot of the database that includes all updates of transactions that commit before *start*(T) and no updates of transactions that commit after *start*(T), except ...
- ... that T sees its own updates.
- If two transactions T_i and T_j are concurrent, then T_i and T_j are not permitted to update the same object.

Snapshot Isolation (SI)

- each transaction T has a start time ($start(T)$) and a commit time ($commit(T)$) - unless it aborts.
- each transacation T “sees” a snapshot of the database that includes all updates of transactions that commit before $start(T)$ and no updates of transactions that commit after $start(T)$, except ...
- ... that T sees its own updates.
- If two transactions T_i and T_j are concurrent, then T_i and T_j are not permitted to update the same object.

Properties of SI

SI provides each transaction with a consistent view of the database, and avoids “lost updates”.

SI vs. Serializability

Consider the following execution history:

$$H = r_1[x]\ r_2[x]\ r_1[y]\ r_2[y]\ w_1[x]\ w_2[y]\ c_1\ c_2$$

- Is this history serializable? In which order can T_1 and T_2 be serialized?
- Is this history SI?

SI vs. Serializability

Consider the following execution history:

$$H = r_1[x]\ r_2[x]\ r_1[y]\ r_2[y]\ w_1[x]\ w_2[y]\ c_1\ c_2$$

- Is this history serializable? In which order can T_1 and T_2 be serialized?
- Is this history SI?

Serializability is stronger than SI

Every serializable history is also SI, but some SI histories are not serializable.

SQL Isolation Levels

Level 3: Serializability

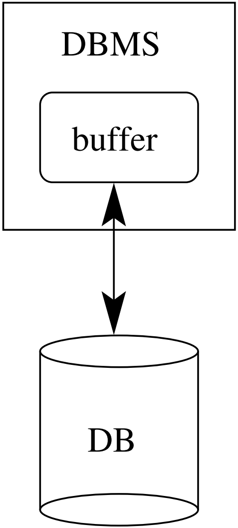
Level 2: Repeatable Read like serializability, but phantoms are possible. Consider:

- T_a orders socks and a bicycle
- T_b reads total value of sock orders, then reads total value of bicycle orders

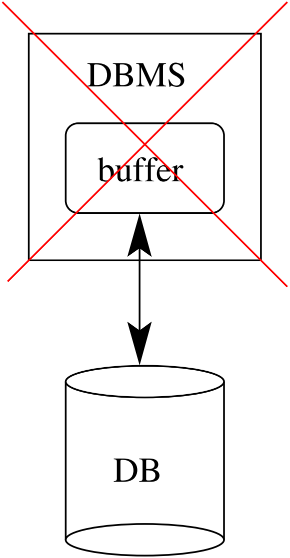
Level 1: Read Committed no ordering guarantees, but transactions will not read uncommitted changes

Level 0: Read Uncommitted here, (almost) anything goes

Handling Failures

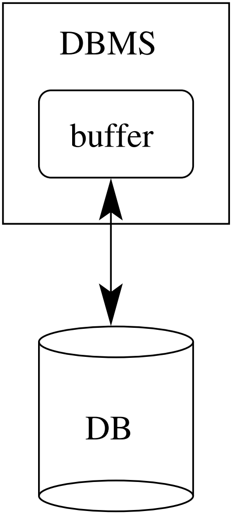


Handling Failures

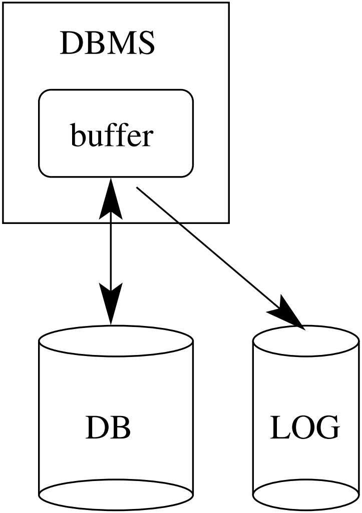


- durability threat: committed updates may be lost
- atomicity threat: uncommitted updates may persist

Write-Ahead Logging (WAL)

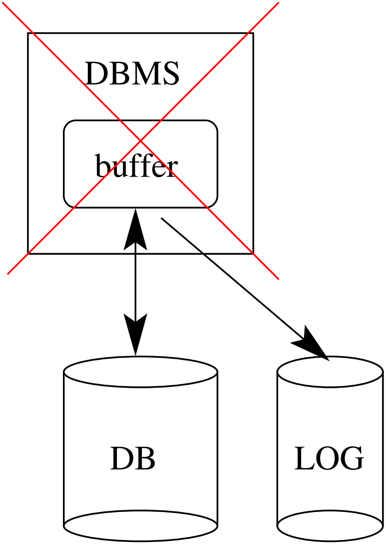


Write-Ahead Logging (WAL)



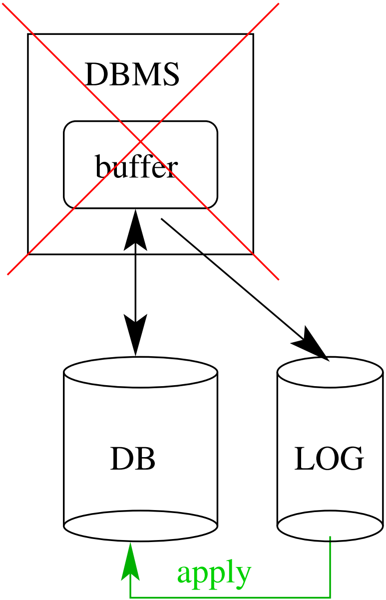
- update the log before updating the DB (ensures unfinished transactions can be undone)
- T's changes logged before T commits (ensures committed transactions will be durable)

Write-Ahead Logging (WAL)



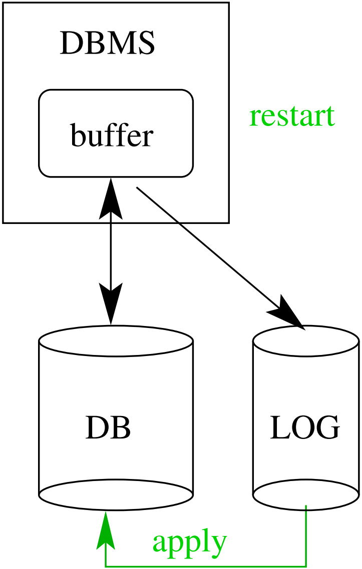
- update the log before updating the DB (ensures unfinished transactions can be undone)
- *T*'s changes logged before *T* commits (ensures committed transactions will be durable)

Write-Ahead Logging (WAL)



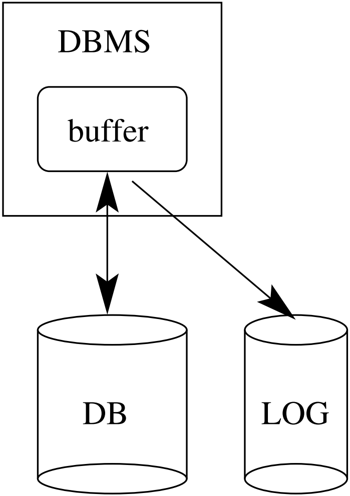
- update the log before updating the DB (ensures unfinished transactions can be undone)
- *T*'s changes logged before *T* commits (ensures committed transactions will be durable)

Write-Ahead Logging (WAL)

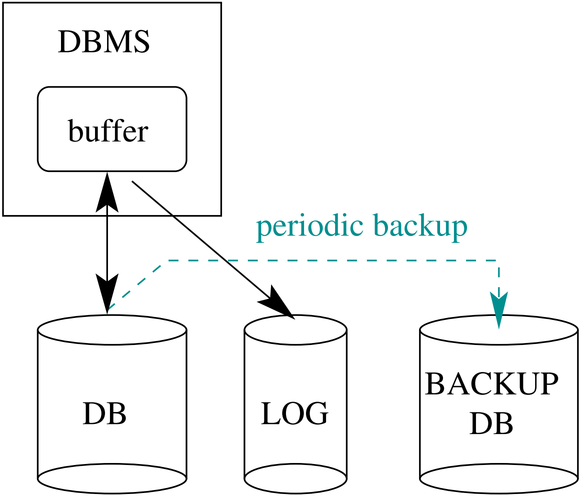


- update the log before updating the DB (ensures unfinished transactions can be undone)
- T's changes logged before T commits (ensures committed transactions will be durable)

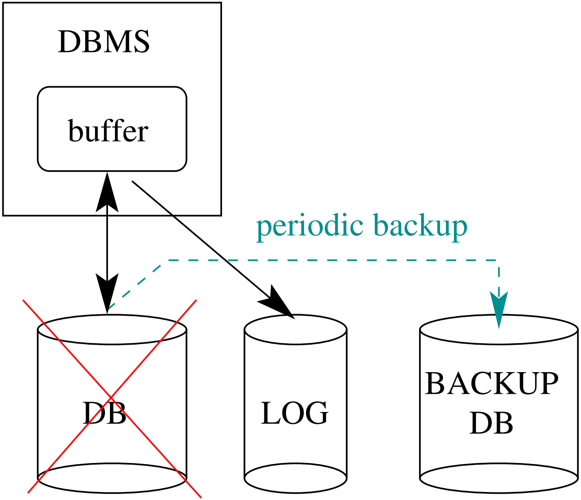
Handling DB Failures



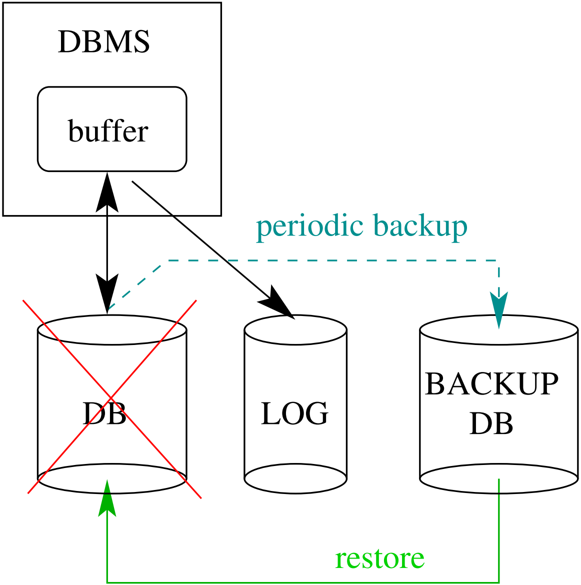
Handling DB Failures



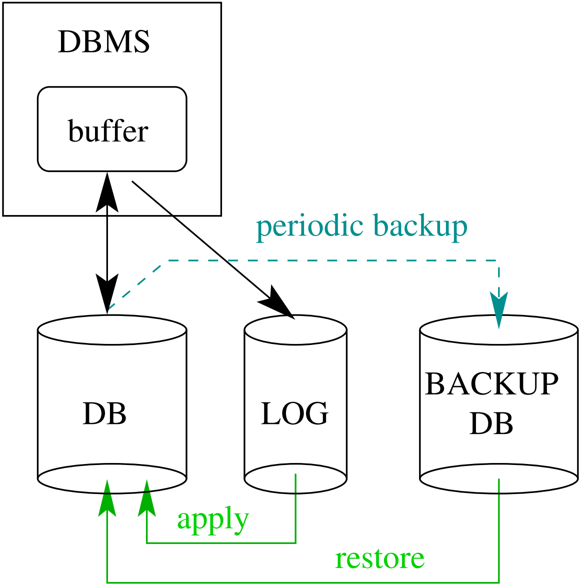
Handling DB Failures



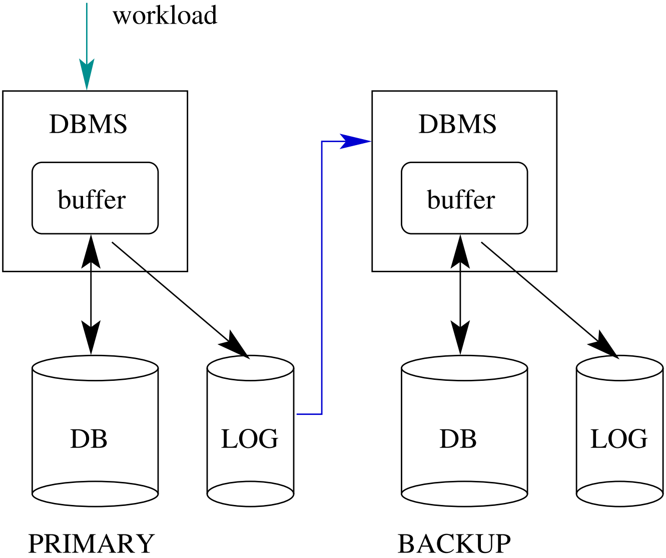
Handling DB Failures



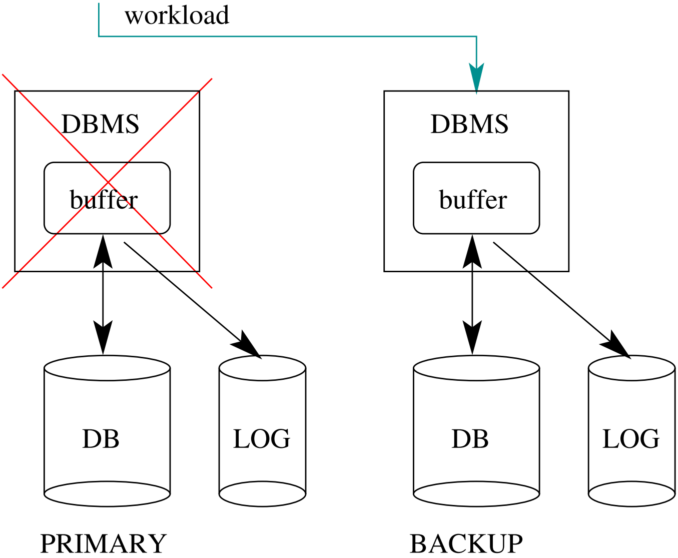
Handling DB Failures



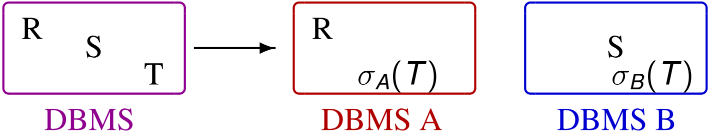
High-Availabilty (HA) DBMS



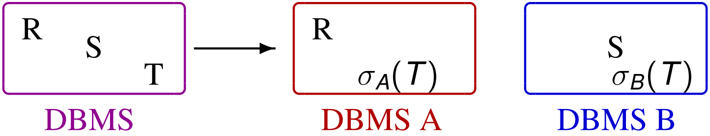
High-Availabilty (HA) DBMS



Data Partitioning

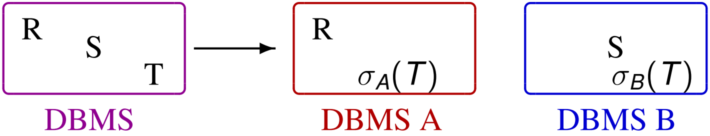


Data Partitioning



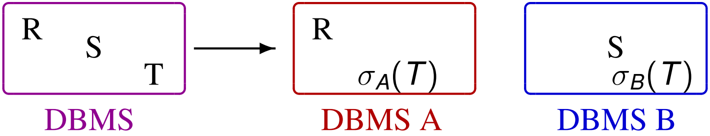
- transactions may span sites (distributed queries, distributed transactions)

Data Partitioning



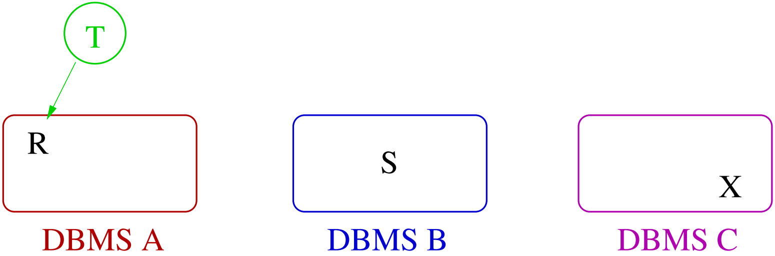
- transactions may span sites (distributed queries, distributed transactions)
- physical design: which data at each site?

Data Partitioning



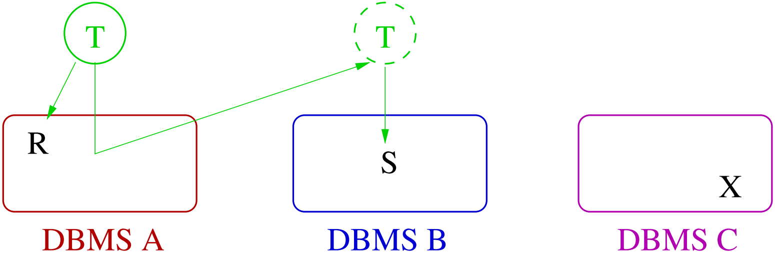
- transactions may span sites (distributed queries, distributed transactions)
- physical design: which data at each site?
- adding/removing sites involves data redistribution

Two Phase Commit (2PC)



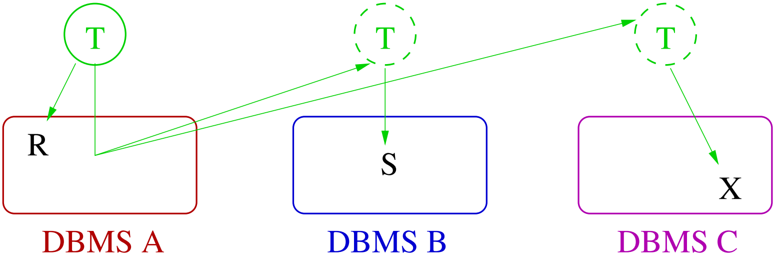
1. `UPDATE R`

Two Phase Commit (2PC)



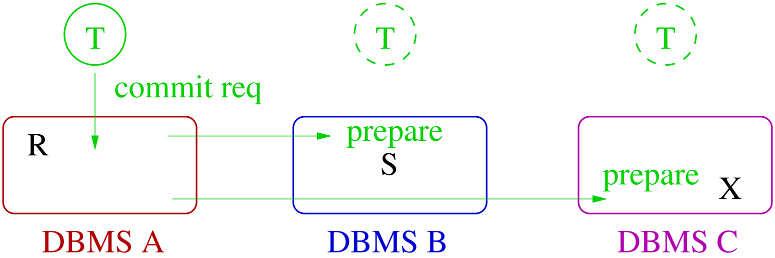
1. UPDATE R
2. UPDATE S

Two Phase Commit (2PC)



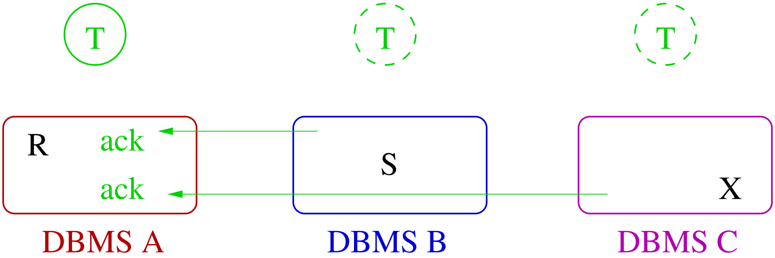
1. `UPDATE R`
2. `UPDATE S`
3. `UPDATE X`

Two Phase Commit (2PC)



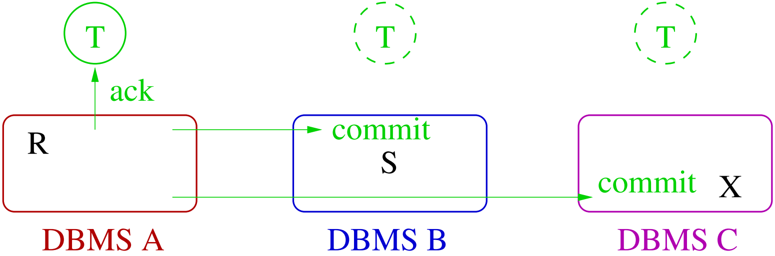
1. `UPDATE R`
2. `UPDATE S`
3. `UPDATE X`
4. `COMMIT`
   - 2PC phase 1

Two Phase Commit (2PC)



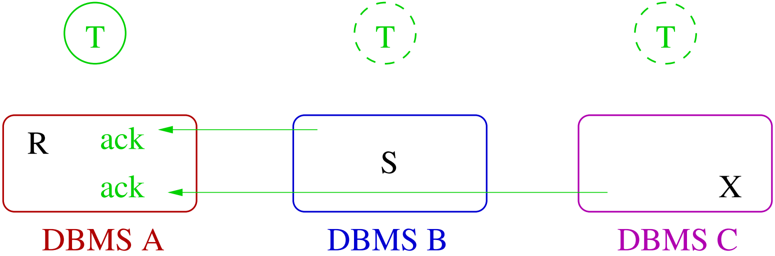
1. `UPDATE R`
2. `UPDATE S`
3. `UPDATE X`
4. `COMMIT`
   - 2PC phase 1

Two Phase Commit (2PC)



1. `UPDATE R`
2. `UPDATE S`
3. `UPDATE X`
4. `COMMIT`
   - 2PC phase 1
   - 2PC phase 2

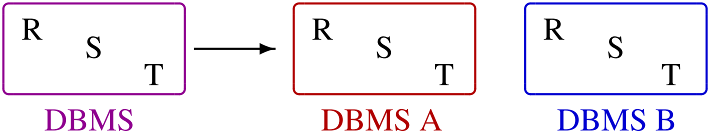
Two Phase Commit (2PC)



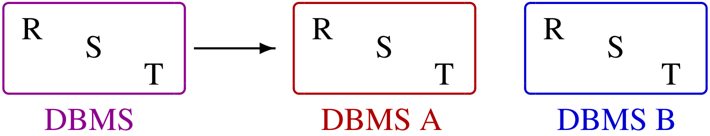
1. UPDATE R
2. UPDATE S
3. UPDATE X
4. COMMIT
    - 2PC phase 1
    - 2PC phase 2

Strict 2PL at each site plus 2PC ensures global serializability.

Data Replication

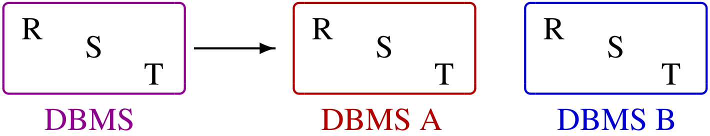


Data Replication



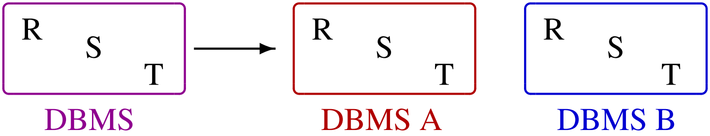
- synchronization: how to keep copies consistent?

Data Replication



- synchronization: how to keep copies consistent?
- replicas are redundant, require extra space

Data Replication



- synchronization: how to keep copies consistent?
- replicas are redundant, require extra space
- simple (though expensive) to add sites, simple to remove sites

1-Copy Serializability (1SR)

- correctness criterion suitable for replicated databases
- system behaves as if there is a single copy of each object on which transactions appear to execute sequentially in some order

Eager Read One, Write All (ROWA) Replication

- to read R, read local replica of R

Eager Read One, Write All (ROWA) Replication

- to read R, read local replica of R
- to update R, update all replicas of R

Eager Read One, Write All (ROWA) Replication

- to read R, read local replica of R
- to update R, update all replicas of R
- each local site has a local concurrency controller

Eager Read One, Write All (ROWA) Replication

- to read *R*, read local replica of *R*
- to update *R*, update all replicas of *R*
- each local site has a local concurrency controller
- use 2PC to atomically commit transaction updates

Eager Read One, Write All (ROWA) Replication

- to read R, read local replica of R
- to update R, update all replicas of R
- each local site has a local concurrency controller
- use 2PC to atomically commit transaction updates

Global Serializability

Local strict two-phase locking $+$ 2PC for commit coordination is sufficient to ensure global 1SR.

Lazy Master/Slave Replication

- one site is designated the master site
- update transactions must run at the master site
- read-only transactions can run at any site
- master site sends updates lazily, in serialization order, to the slave sites
- slaves apply the updates in the order in which they are received
- 2PC is not needed, as all transactions are single-site

Global Serializability

Global 1SR is ensured (why?), but read-only transactions may see stale data.

CAP

Consistency: serializability (or SI)

Availability: nodes that are up should eventually respond to requests

Partition-Tolerance: system should continue to operate even if it partitions

CAP

Consistency: serializability (or SI)

Availability: nodes that are up should eventually respond to requests

Partition-Tolerance: system should continue to operate even if it partitions

Brewer's CAP Conjecture (PODC 2000)

It is impossible build a [distributed database] system that provides consistency, availability, and partition-tolerance.

Distributed DB and CAP

Partitioned Data: ensures consistency but availability suffers in case of site failures or partitions

Distributed DB and CAP

Partitioned Data: ensures consistency but availability suffers in case of site failures or partitions

Eager ROWA Replication: ensures consistency but partitions can block 2PC and node failures prevent updates, hurting availability

Distributed DB and CAP

Partitioned Data: ensures consistency but availability suffers in case of site failures or partitions

Eager ROWA Replication: ensures consistency but partitions can block 2PC and node failures prevent updates, hurting availability

Lazy Master/Slave Replications: ensures (weak) CAP for read-only transactions but partitions or master failure can prevent all updates, hurting availability

Views

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND
       LibraryId = 'UW'
```

Views

Views are named queries that can be used much like regular tables.

Materialized Views

- materialized views are views for which the result of the underlying view query has been computed and stored
- materialized views may be used (in place of the base tables) to answer some queries
- one challenge is synchronizing materialized views with the underlying tables as those tables are update

Full replication is a special case of view materialization.

Views and Updates

```
CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

Views and Updates

```
CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

- Changes (INSERT, DELETE, UPDATE) to Books may change the result of the query that defines CSBooks.

Views and Updates

```
CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

- Changes (INSERT, DELETE, UPDATE) to Books may change the result of the query that defines CSBooks.
- Changes to Holdings may change the result of the query that defines UWHoldings.

Views and Updates

```
CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

- Changes (INSERT, DELETE, UPDATE) to Books may change the result of the query that defines CSBooks.
- Changes to Holdings may change the result of the query that defines UWHoldings.

Update Relevance

An update is relevant to a view if that update could change the result of the view's underlying query.

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

- immediate: view is updated within the transaction that updates the underlying table

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

- immediate: view is updated within the transaction that updates the underlying table
- deferred: view updated occurs after the underlying table is updated

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

- immediate: view is updated within the transaction that updates the underlying table
- deferred: view updated occurs after the underlying table is updated

mechanism: how is the materialized view updated?

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

- immediate: view is updated within the transaction that updates the underlying table
- deferred: view updated occurs after the underlying table is updated

mechanism: how is the materialized view updated?

- full refresh: recompute the view after the underlying table is updated

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

- immediate: view is updated within the transaction that updates the underlying table
- deferred: view updated occurs after the underlying table is updated

mechanism: how is the materialized view updated?

- full refresh: recompute the view after the underlying table is updated
- incremental refresh: compute the view changes that result from the update, and apply them to the old materialized view

Incremental Refresh

```
Books (BookId, Title, Author, Subject, Year)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'
```

Suppose tuple *t* is inserted into `Books`. Incremental maintenance of `CSBooks` involves:

Incremental Refresh

```
Books (BookId, Title, Author, Subject, Year)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'
```

Suppose tuple *t* is inserted into `Books`. Incremental maintenance of `CSBooks` involves:

1. test whether *t*.`Subject = 'CS'`

Incremental Refresh

```
Books (BookId, Title, Author, Subject, Year)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = 'CS'
```

Suppose tuple *t* is inserted into `Books`. Incremental maintenance of `CSBooks` involves:

1. test whether *t*.`Subject = 'CS'`
2. if so, insert *t* into `CSBooks`

Incremental Refresh (cont'd)

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)
CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

Suppose tuple *t* is inserted into `Holdings`. Incremental maintenance of `UWHoldings` involves:

Incremental Refresh (cont'd)

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)
CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

Suppose tuple *t* is inserted into `Holdings`. Incremental maintenance of `UWHoldings` involves:

1. test whether *t*.`LibraryId = 'UW'`

Incremental Refresh (cont'd)

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)
CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

Suppose tuple *t* is inserted into `Holdings`. Incremental maintenance of `UWHoldings` involves:

1. test whether *t*.`LibraryId = 'UW'`
2. join *t* with `Books` on *t*.`BookId = Books.BookId`

Incremental Refresh (cont'd)

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)
CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

Suppose tuple *t* is inserted into `Holdings`. Incremental maintenance of `UWHoldings` involves:

1. test whether *t*.`LibraryId = 'UW'`
2. join *t* with `Books` on *t*.`BookId = Books.BookId`
3. insert the resulting `Title` into `UWHOldings`

Incremental Refresh (cont'd)

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)
CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = 'UW'
```

Suppose tuple *t* is inserted into `Holdings`. Incremental maintenance of `UWHoldings` involves:

1. test whether *t*.`LibraryId = 'UW'`
2. join *t* with `Books` on *t*.`BookId = Books.BookId`
3. insert the resulting `Title` into `UWHOldings`

Self-Maintainability

`UWHoldings` is not self-maintainable wrt inserts into `Holdings`.

Using Materialized Views

- user-visible
  - MV is defined and named by an application or administrator
  - application may refer to the MV in queries
  - application or administrator defines synchronization policies
- transparent
  - MVs are defined and created by the system
  - applications do not refer directly to the MVs in queries
  - query optimizer may rewrite user queries to use MVs

Query Caching

- materialize query results and use them to answer subsequent queries more quickly

Query Caching

- materialize query results and use them to answer subsequent queries more quickly
- a special case of view materialization:
    - dynamic set of materialized queries

Query Caching

- materialize query results and use them to answer subsequent queries more quickly
- a special case of view materialization:
  - dynamic set of materialized queries
  - transparent to applications
    - exact matching based on query text
    - more general or partial matching

Query Caching

- materialize query results and use them to answer subsequent queries more quickly
- a special case of view materialization:
  - dynamic set of materialized queries
  - transparent to applications
    - exact matching based on query text
    - more general or partial matching
  - sychronization
    - incremental refresh
    - invalidation