Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Use multiversion 2-phase locking
  - Give logical “snapshot” of database state to read only transaction
    - Reads performed on snapshot
  - Update (read-write) transactions use normal locking
  - Works well, but how does system know a transaction is read only?
- Solution 2 (partial): Give snapshot of database state to every transaction
  - Reads performed on snapshot
  - Use 2-phase locking on updated data items
  - Problem: variety of anomalies such as lost update can result
  - Better solution: snapshot isolation level (next slide)
### Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible to T1
  - Writes of T1 complete when it commits
- **First-commmitter-wins rule:**
  - Commits only if no other concurrent transaction has already written data that T1 intends to write.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W(Y := 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(X) → 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(Y) → 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(X:=2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(Z:=3)</td>
<td>Commit</td>
</tr>
</tbody>
</table>

 Concurrent updates not visible
 Own updates are visible
 Not first-commmitter of X
 Serialization error, T2 is rolled back

### Snapshot Read

- Concurrent updates invisible to snapshot read

\[
X_0 = 100, \ Y_0 = 0
\]

<table>
<thead>
<tr>
<th>T_1 deposits 50 in Y</th>
<th>T_2 withdraws 50 from X</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r_1(X_0, 100)</td>
<td>\r_2(Y_0, 0)</td>
</tr>
<tr>
<td>\r_1(Y_0, 0)</td>
<td>\r_2(X_0, 100)</td>
</tr>
<tr>
<td>\w_1(Y_1, 50)</td>
<td>\r_2(Y_0, 0) (update by T_2 not seen)</td>
</tr>
<tr>
<td>r_1(X_0, 100) (update by T_2 not seen)</td>
<td>\r_2(X_0, 100)</td>
</tr>
<tr>
<td>r_1(Y_1, 50) (can see its own updates)</td>
<td>\w_2(X_2, 50)</td>
</tr>
</tbody>
</table>

\[
X_2 = 50, \ Y_1 = 50
\]
**Snapshot Write:** First Committer Wins

<table>
<thead>
<tr>
<th></th>
<th>$X_0 = 100$</th>
<th>$X_1 = 150$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$ deposits 50 in $X$</td>
<td>$T_2$ withdraws 50 from $X$</td>
<td></td>
</tr>
<tr>
<td>$r_1(X_0, 100)$</td>
<td>$r_2(X_0, 100)$</td>
<td></td>
</tr>
<tr>
<td>$w_1(X_1, 150)$</td>
<td>$w_2(X_2, 50)$</td>
<td></td>
</tr>
<tr>
<td>commit$_1$</td>
<td>commit$_2$ (serialization error $T_2$ is rolled back)</td>
<td></td>
</tr>
</tbody>
</table>

- **Variant:** “**First-updater-wins**
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

**Benefits of SI**

- Reads are *never* blocked,
  - and also don’t block other txns activities
- Performance similar to Read Committed
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - No lost update
    - I.e., update made by a transaction is overwritten by another transaction that did not see the update
  - No non-repeatable read
    - I.e., if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated
Snapshot Isolation

- Example of problem with SI
  - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ?
    - if both transactions start at the same time, with snapshot isolation: A = ?? , B = ??
- Called skew write
- Skew also occurs with inserts
  - E.g.,:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transactions can both create order with same number
      - Is an example of phantom phenomenon


table

<table>
<thead>
<tr>
<th>$T_i$</th>
<th>$T_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>$A=B$</td>
<td>$A=B$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

Snapshot Isolation Anomalies

- SI breaks serializability when transactions modify different items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
  - But problems do occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot
Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI):** extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - Where \( T_i \) writes a data item \( Q \), \( T_j \) reads an earlier version of \( Q \), but \( T_j \) is serialized after \( T_i \)
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
  - Ensures serializability
  - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
  - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring true serializability

SI Implementations

- Snapshot isolation supported by many databases
  - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
  - Isolation level can be set to snapshot isolation
- Oracle implements “first updater wins” rule (variant of “first committer wins”)
  - Concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
- **Warning:** *even if isolation level is set to serializable, Oracle actually uses snapshot isolation*
  - Old versions of PostgreSQL prior to 9.1 did this too
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution
Working Around SI Anomalies

- Can work around SI anomalies for specific queries by using `select .. for update` (supported e.g. in Oracle)
  - Example
    - `select max(orderno) from orders for update`
    - read value into local variable maxorder
    - insert into orders (maxorder+1, …)
- `select for update (SFU) clause` treats all data read by the query as if it were also updated, preventing concurrent updates
- Can be added to queries to ensure serializability in many applications
  - Does not handle phantom phenomenon/predicate reads though

Spark: Another Implementation of this Programming Model

BDAS Stack

- Spark Streaming
- BlinkDB
- Sample Clean
- Spark SQL
- SparkR
- GraphX
- KestoneML
- MLlib
- Velox

Spark Core

- Mesos
- Succinct
- Tachyon

Storage

- Hadoop Yarn
- HDFS, S3, Ceph, …

Res. Mgmt.

3rd party
Spark Applications

Spark

Unifies batch, interactive, streaming computations
Easy to build sophisticated applications
• Support iterative, graph-parallel algorithms
• Powerful APIs in Scala, Python, Java, R

Spark 101

• Apache Spark
  – Open Source
  – Extensive developer community
  – Growing commercial use
• Somewhat heavy to set up
  – First, you need a cloud…
Spark: Key Differences from Map-Reduce

• **Lazy Execution**
  – Lambda-functions – pass the function to compute the result
  – Only compute when needed
  – *Similar to pipelining*

• More control over execution
  – Beyond simple key/value mappings

• Syntactically seems complex
  – But the ideas are the same

Creating a Data Object

```python
>>> sc
<pyspark.context.SparkContext object at 0x10ea7d4d0>
>>> pagecounts = sc.textFile("data/pagecounts")
>>> pagecounts
MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:2
• *Assume data/pagecounts is pageviews of Wikipedia pages*
```
Viewing Data
(first 10 records)

```python
>>> pagecounts.take(10)
...
[u'20090505-000000 aa.b ?71G4Bo1cAdWyg 1 14463', u'20090505-000000 aa.b Special:Statistics 1 840', u'20090505-000000 aa.b Special:Whatlinkshere/MediaWiki:Returnto 1 1019', u'20090505-000000 aa.b Wikibooks:About 1 15719', u'20090505-000000 aa ?14mFX1ildVnBc 1 13205', u'20090505-000000 aa ?53A%2FuYP3FnKM 1 13207', u'20090505-000000 aa ?93HqrnFc%2EiqRU 1 13199', u'20090505-000000 aa ?95iZ%2Fjuimv31g 1 13201', u'20090505-000000 aa File:Wikinews-logo.svg 1 8357', u'20090505-000000 aa Main_Page 2 9980']
```

Prettier:

```python
>>> for x in pagecounts.take(10):
...    print x
...
20090505-000000 aa.b ?71G4Bo1cAdWyg 1 14463
20090505-000000 aa.b Special:Statistics 1 840
20090505-000000 aa.b Special:Whatlinkshere/MediaWiki:Returnto 1 1019
20090505-000000 aa.b Wikibooks:About 1 15719
20090505-000000 aa ?14mFX1ildVnBc 1 13205
20090505-000000 aa ?53A%2FuYP3FnKM 1 13207
20090505-000000 aa ?93HqrnFc%2EiqRU 1 13199
20090505-000000 aa ?95iZ%2Fjuimv31g 1 13201
20090505-000000 aa File:Wikinews-logo.svg 1 8357
20090505-000000 aa Main_Page 2 9980
```
Caching Results

```python
>>> pagecounts.count()
• May take a long time
>>> enPages = pagecounts.filter(lambda x: x.split(" ")[1] == "en").cache()
• doesn’t actually do anything
>>> enPages.count()
• slow the first time, fast in later calls
```

Histogram of page views

• First, divide the data
  ```python
  >>> enTuples = enPages.map(lambda x: x.split(" "))
  ```
• And create a count for each date
  ```python
  >>> enKeyValuePairs = enTuples.map(lambda x: (x[0][:8], int(x[3])))
  ```
• Then combine
  ```python
  >>> enKeyValuePairs.reduceByKey(lambda x, y: x + y, 1).collect()
  `[(u'20090507', 6175726), (u'20090505', 7076855)]`
```
Single command to do it all (and only return where >200k)

```python
>>> enPages.map(lambda x: x.split(" ")).
map(lambda x: (x[2],int(x[3]))).
reduceByKey(lambda x, y: x + y, 40).
filter(lambda x: x[1] > 200000).
map(lambda x: (x[1], x[0])).collect()
```

```
[(451126, u'Main_Page'), (1066734, u'404_error/'), (468159, u'Special:Search')]
```

Summary

- Map/Reduce framework enables high throughput on large data
  - Provided you can frame the solution as a map / reduce
  - Simple, but somewhat different, programming model
- Distributed data, distributed processing
  - Distribution handled by underlying software
- Multiple implementations available
  - Hadoop, Spark, others…
Bloom Filters

- A **bloom filter** is a probabilistic data structure used to check membership of a value in a set
  - May return true (with low probability) even if an element is not present
  - But never returns false if an element is present
  - Used to filter out irrelevant sets
- Key data structure is a single bitmap
  - For a set with $n$ elements, typical bitmap size is $10n$
- Uses multiple independent hash functions
- With a single hash function $h()$ with range=number of bits in bitmap:
  - For each element $s$ in set $S$ compute $h(s)$ and set bit $h(s)$
  - To query an element $v$, compute $h(v)$, and check if bit $h(v)$ is set
- Problem with single hash function: significant chance of false positive due to hash collision
  - 10% chance with $10n$ bits

Bloom Filters (Cont.)

- Key idea of Bloom filter: reduce false positives by use multiple hash functions $h_i()$ for $i = 1..k$
  - For each element $s$ in set $S$ for each $i$ compute $h_i(s)$ and set bit $h_i(s)$
  - To query an element $v$, for each $i$ compute $h_i(v)$, and check if bit $h_i(v)$ is set
    - If bit $h_i(v)$ is set for every $i$ then report $v$ as present in set
    - Else report $v$ as absent
  - With $10n$ bits, and $k = 7$, false positive rate reduces to 1% instead of 10% with $k = 1$