CS848 Paper Presentation
Building a Database on S3

Brantner, Florescu, Graf, Kossmann, Kraska

SIGMOD 2008

Presented by Jason Ho

David R. Cheriton School of Computer Science

University of Waterloo

15 March 2010

Outline

- ❑Background
- ❑S3
- ❑SQS
- ❑Using S3 As a Disk
- ❑Basic Commit Protocols
- ❑Experiment
- ❑Conclusion
- ❑Question

Background

❑Utility computing (aka cloud computing) provides basic hardware/software ingredients as commodity with low unit cost, AND it provides scalability, availability, and constant response times for every client.

❑S3 as a popular representative of utility service today which is also part of Amazon Web Services (AWS) including SQS, EC2.

Simple Storage System (S3)

❑Provide infinite storage for S3 object (1B to 5GB)

❑S3 objects can be identified by an URI

❑Provide SOAP or REST-based interface to access S3 objects:

- *get*(uri) : returns a object with given URI
- *put(*uri, bytestream) : Writes bytestream to URI object
- *get-if-modified-since*(uri, timestamp) : Given the URI, gets new version of object since specific timestamp

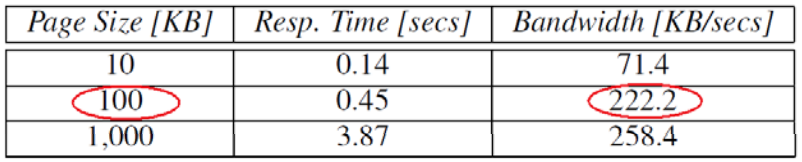
❑Each S3 object is associated with a bucket. Also, bucket can be a unit of security

Simple Storage System (S3)

- Cost for S3:
    - $0.15 to store 1GB data in S3 per month
    - $0.1 per 10,000 get requests
    - $0.1 per 1,000 put requests
    - $0.18 per GB of network bandwidth consumed
    - Thus, S3 is good for persistent storage
- S3 latency issue:
    - Reading from S3 takes at least 100 msecs which is 2 to 3 times longer than reading from local disk
    - Writing to S3 takes 3 times long as reading data

Simple Storage System (S3)

- Acceptable bandwidth is feasible if data are read in relatively large chunks of 100KB and more.



Page Size [KB]	*Resp. Time [secs]*	*Bandwidth [KB/secs]*
10	0.14	71.4
100	0.45	222.2
1,000	3.87	258.4

Table 1: Resp. Time, Bandwidth of S3, Vary Page Size

- S3 implementation detail is not published:
  - Stored data is replicated across several data centers
  - Availability guaranteed; If one data center fails, data replica in another data center is used for read and update

Simple Queuing System (SQS)

❑Allow user to manage infinite number of queues with infinite capacity (virtually)

❑SQS Queue is a stack of message and it can be referred by URI

❑Supports sending and receiving message via a HTTP or REST-based interface:

- *createQueue*(uri) : To create a new queue with given URI
- *send*(uri, msg) : To send a message to queue identified by given URI. Returns the ID of the message
- *receive*(uri, num, timeout) : To receive num messages from the top of URI queue. *The returned messages are locked in timeout period*.
- *delete*(uri, msg-id) : To delete a message with msg-id in a queue identified by given URI
- *addGrant*(uri, usr) : To grant a user to access a queue

❑Each message is identified by a unique ID

Simple Queuing System (SQS)

- ❑Cost for SQS; $0.01 to send 1,000 messages
- ❑Network bandwidth costs at least $0.1 per GB of data transferred
- ❑SQS implementation detail is not published:
  - ❑ Messages of queues are replicated in on many machines in different data centers.
  - ❑ When clients initiate request, theyshould not be blocked by system failure and other clients and receive the request results in constant time.

Using S3 As a Disk

- Client-Server Architecture of an S3 database:
  - Similar to a distributed shared-disk database system
  - Clients retrieve pages from S3 based on pages' URI, buffer the pages locally, update them, and write them back to S3
- Page: the unit of transfer records between clients and S3
- Client consisting "Page Manager", "Record Manager", and "Application"

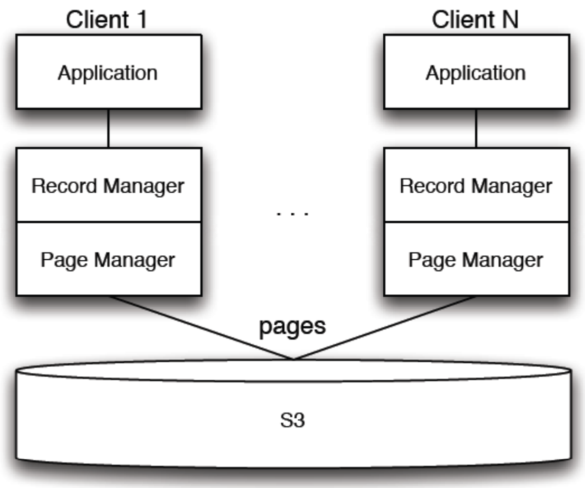


Figure 1: Shared-disk Architecture

Using S3 As a Disk : Record Manager

- To manage records:
    - A record is composed of a unique key and a payload data.
    - Physically, each record is stored in exactly 1 page which in turn is stored as a single object in S3.
    - Logically, each record is part of a collection (e.g., table).
    - A collection is implemented as a bucket in S3 and all the pages that store records of that collection are stored as S3 object in that bucket.
    - A collection can be identified by URI.
- Provides funtions to access/modify records in collections:
    - *Create(key, payload, uri)* : To create a new record with a payload data in a collection identified by URI
    - *Read(key, uri)* : To read the payload data of a record with given key locating in the collection with uri
    - *Update(key, payload, uri)* : To update the payload data of a record.
    - *Delete(key, uri)* : To delete a record with given key in the collection with given uri
    - *Scan(uri)* : To scan through all records of a collection of given uri.

Using S3 As a Disk : Page Manager

- Implement buffer pool for S3 pages
- Support reading pages from S3, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated
- Create a new page in S3
- *Commit* and *abort* transactions
- The pages in the buffer pool can be marked as either *unmodified, modified*, or *new.*
    - When application commits a transaction, all the updates will be propagated to S3 and all the affected pages are marked as unmodified in the client's buffer pool
    - When application aborts a transaction, all pages marked modified or new are discarded from the buffer pool

Using S3 As a Disk : B-Tree Indexes

- ❑B-tree can be implemented on top of the page manager:
  - ❑ Pages of root nodes and intermediate nodes of the B-tree contains (*key*, *uri*) pairs, where uri refers to the appropriate page at the next lower level
  - ❑ Leaf pages of a primary index contains (*key, payload data*) pairs
  - ❑ Leaf page of secondary index contains (*search key, record key*) pairs
- ❑Nodes at the same level are linked by pointer
- ❑B-tree is identified by the URI of the root node

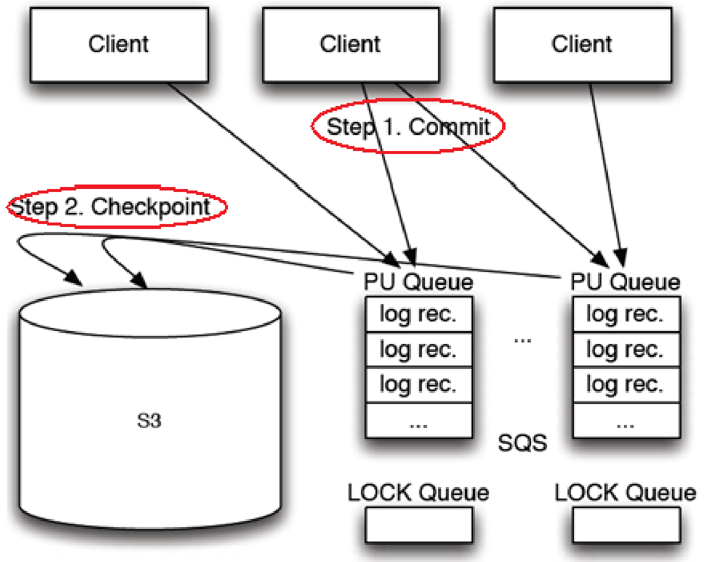
Using S3 As a Disk : Logging

- A log record is always associated to a collection.
- 3 types of log record:
  - (*insert, key, payload*)
  - (*delete, key*)
  - (*update, key, afterimage*)
- These log records are idempotent

Basic Commit Protocols

- Problem: With the S3 database system previous described, the updates of one client can be overwritten by another client even if the 2 clients update different records.
- Why? The unit of transfer is a page rather than a single record.
- Basic commit protocol of how client commits updates consisting of 2 steps:
  - *Commit*: client generates log records for all the updates committed and sends them to SQS
  - *Checkpoint*: log records are applied to the pages storing on S3

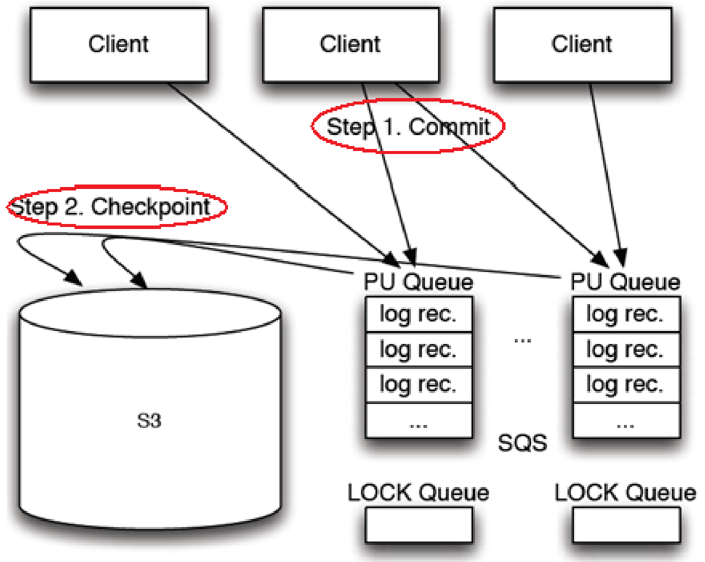
(CONTINUE)



Basic Commit Protocol

Basic Commit Protocols

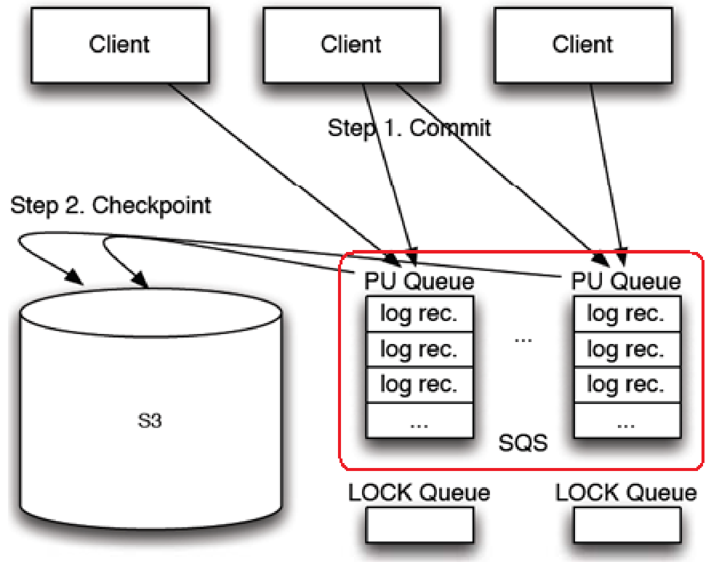
- ❑Commit step is carried out in constant time assuming a constant number of messages sent per commit
- ❑Checkpoint step can be carried out asynchronously and outside of the execution of a client application, and thus users are not blocked by the checkpoint step.
- ❑Resilient to failures: When client fails in the commit step, the client resends all log records when it restarts.
- ❑Violating atomicity: The client never come back from failure and loses uncommitted log records.
- ❑Eventual Consistency: all update will become visible to everyone eventually.



Basic Commit Protocol

Basic Commit Protocols : PU Queues

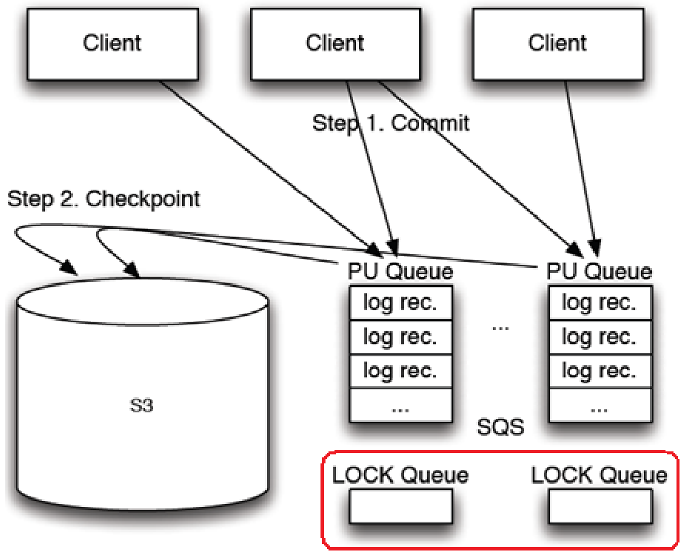
- ❑Client propagate log records to Pending Update queues (PU queues)
- ❑Each B-tree has one PU queue
- ❑One PU queue is associated to each leaf node of a primary B-tree of a collection



Basic Commit Protocol

Basic Commit Protocols : Checkpoint Protocol for Data Pages

- ❑Input of a checkpoint step is a PU queue
- ❑To ensure no other clients can carry out a checkpoint on the same PU queue currently, a LOCK queue is associated to each PU queue.

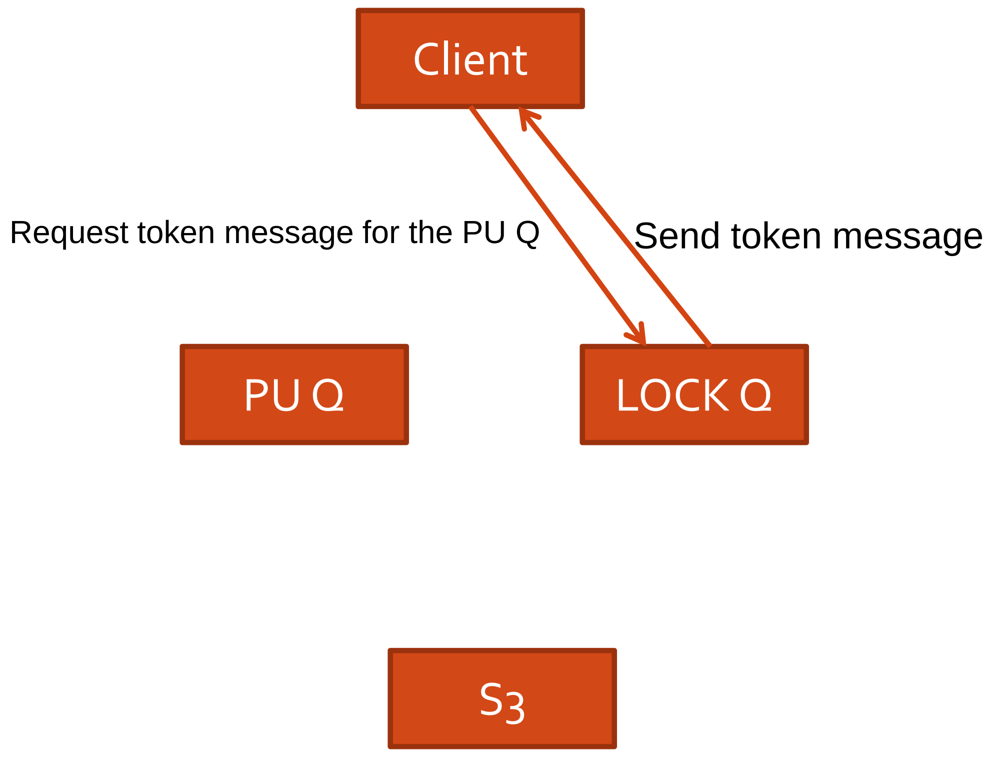


Basic Commit Protocol

(CONTINUE)

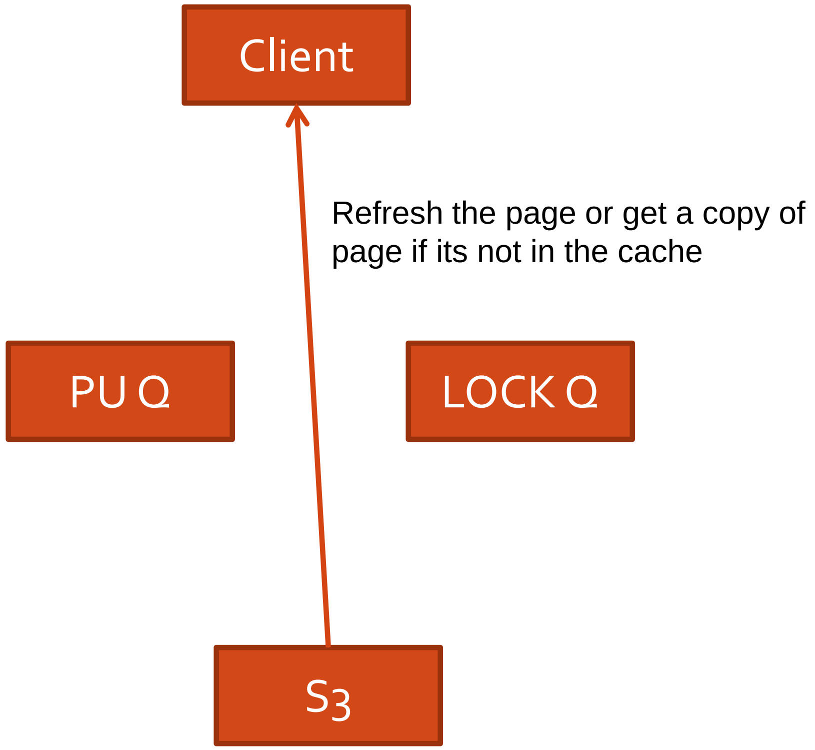
Basic Commit Protocols : Checkpoint Protocol for Data Pages

1. Request to obtain token message from the LOCK queue.
   - If the token message is returned, set the timeout.
   - If the token message is NOT return, terminate
   - Set the timeout period



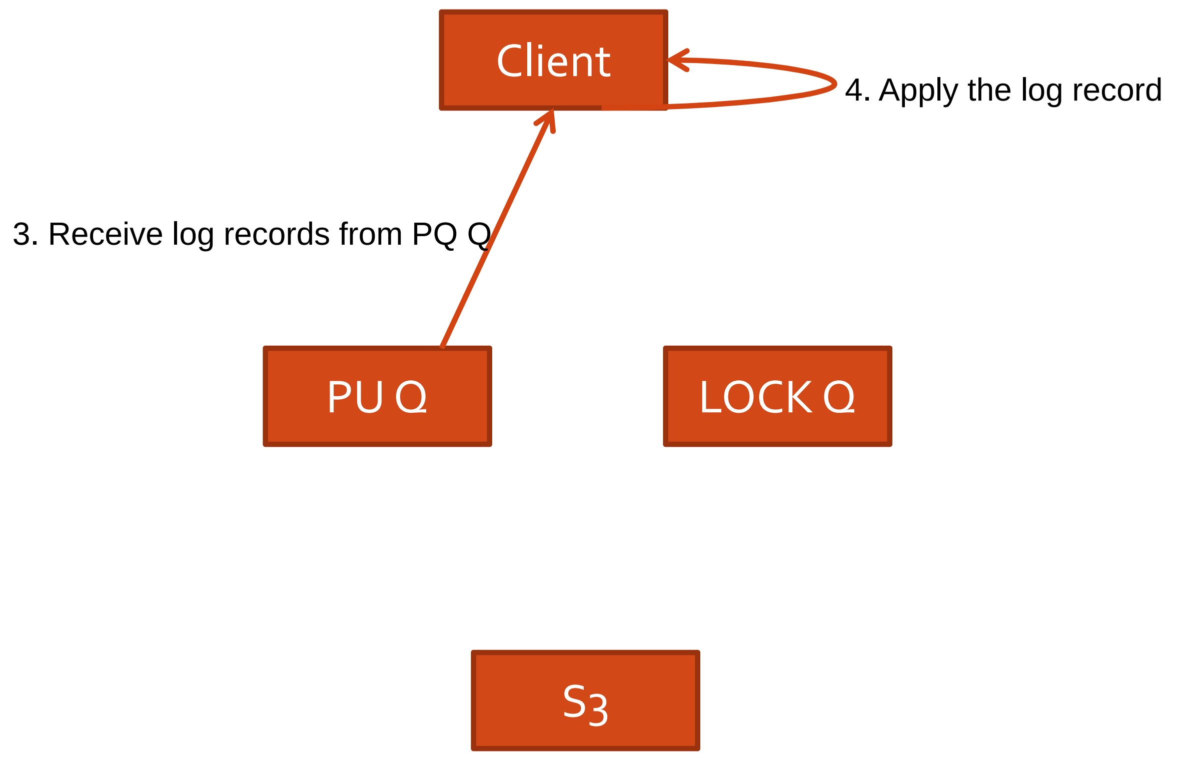
Basic Commit Protocols : Checkpoint Protocol for Data Pages

2. If the page is cached, refresh the cached copy. If the page is not cached, get a copy of the page from S3


Client
Refresh the page or get a copy of page if its not in the cache
PU Q
LOCK Q
S3

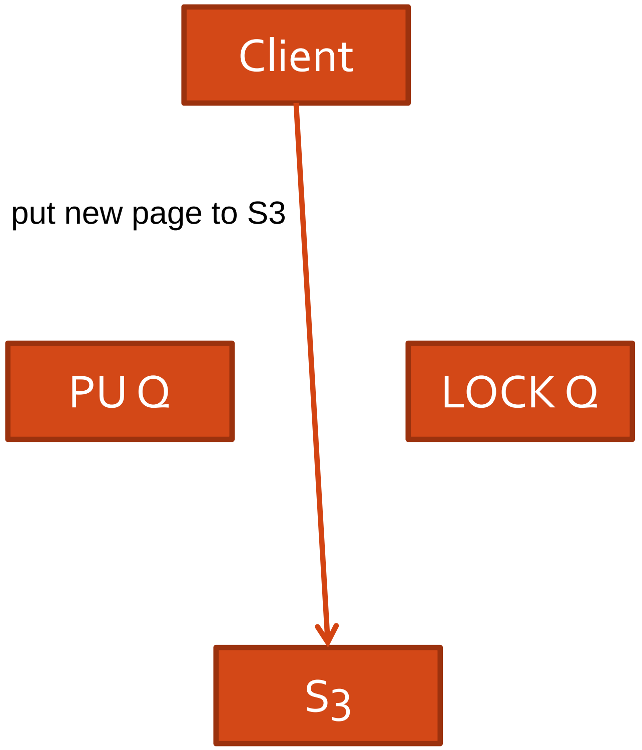
Basic Commit Protocols : Checkpoint Protocol for Data Pages

3. Receive as many log records from the PU queue as possible.
4. Apply the log records to the cached copy of the page at the client.



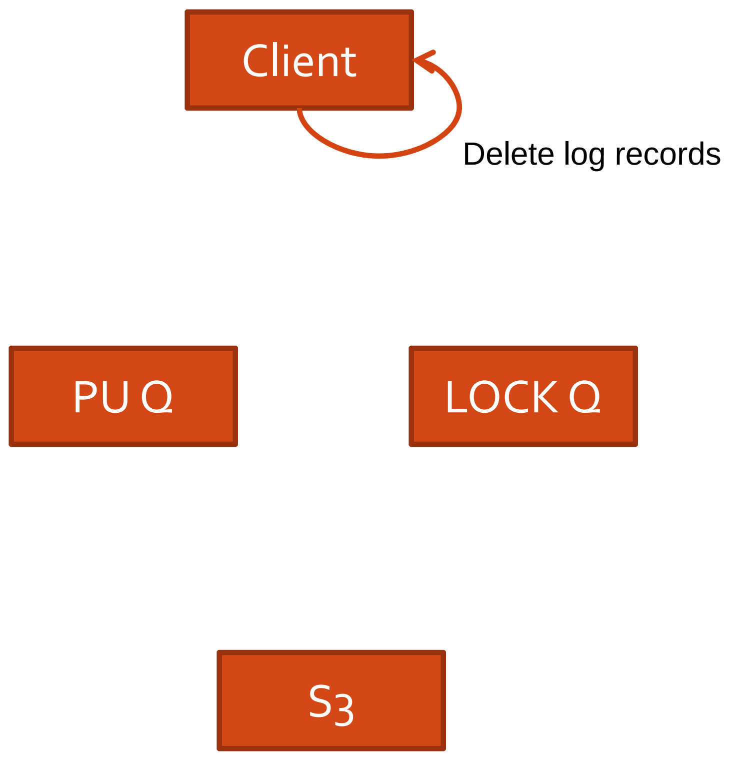
Basic Commit Protocols : Checkpoint Protocol for Data Pages

5. If timeout is not expired, put the new version of the page to S3



Basic Commit Protocols : Checkpoint Protocol for Data Pages

6. If the above steps are carried out successfully within timeout, delete all the log records received in Step 3 from PU queue



Basic Commit Protocols : Checkpoint Protocol for B-Tree

❑Checkpoint on B-tree is complicated because it involves more pages in the process:

1. Obtain token from Lock Queue and set timeout period
2. Receive log records form the PU Queue
3. Sort the log records by Key
4. Find leaf node in B-tree for first log record, and refresh the leaf node from S3.
5. Apply all log records to that leaf node
6. If timeout is not expired, put new version of node to S3
7. If timeout is not expired, delete log records
8. If there are still log records need to be process, go to step 4. Otherwise, terminate.

Basic Commit Protocols : Checkpoint Strategies

- ❑Checkpoint on a page can be carried out by *Reader*, *Writer*, *Watchdog*, or *Owner*.
- ❑If a *Writer* initiates a checkpoint using the following condition:
  - ❑ Each data page records the timestamp of the last checkpoint
  - ❑ When a client commits a log record to a data page or B-tree in S3, the client computes the difference between its wallclock time and the timestamp recorded
- ❑If the absolute value of this difference is bigger than a certain threshold (checkpoint interval), the writer carries out a checkpoint.
- ❑Flaw in writer-only strategy: it is possible that a page which is updated once and then never again is never checkpointed. As a result, the update never becomes visible.
- ❑Solution: Readers initiate checkpoints if they see a page whose last checkpoint was a long time ago.

Transactional Properties

❑Additional transactional properties such as *atomicity* and *client-side consistency* can be achieved at as low as possible cost without sacrificing the basic utility computing principles.

Transactional Properties : Atomicity

- Problem: the basic commit protocol previous mentioned cannot achieve full atomicity.
- Solution: The client commits log records to a ATOMIC queues rather than PU queue, and each log record is annotated with an **ID** which uniquely identify a transaction on the client
- When a client fails and comes back:
  - In the ATOMIC queue, log records with the matching **ID** as the commit record are propagated to PU queue, and then deleted from the ATOMIC queue.
  - These ones with no matching **ID** are deleted immediately from the ATOMIC queue and not propagated to PU queues

Transactional Properties : Consistency Levels

❑Client-side consistency models:

- Monotonic Reads : If a client read the value of a data x, any successive read operation on x by that client will always return the same value or a more recent value
- Monotonic Writes : A write operation by a client on data x is completed before any successive write operation on x by the same client
- Read your writes : The effect of a write on data x by a client will always be seen by a successive read operation on x by the same client
- Write follows read : A write on data x following a previous read on data x by the same client, is guaranteed to take place on the same or a more recent value of x that was read.

Experiment

- ❑The experience is carried out in 1 baseline and 3 level of consistency:
- ❑Naïve approach as baseline:
  - ❑ write all dirty pages to S3
  - ❑ Is subject to lost updates, and thus it holds **lowest level of consistency**.
- ❑LEVEL OF CONSISTENCY DESCRIPTIONS:
  - Basic : The protocol only supports the basic commit protocol (Commit and Checkpoint steps) and supports eventual consistency
  - Monotonicity : On top of Basic and supports full clients side consistency models.
  - Atomicity : Above Monotonicity and Basic with atomicity. It holds **highest level of consistency.**
- ❑For the 4 variants, the implementation of read, write, create, index probe and abort operations in the record manager, page manager, and B-tree index are identical.
- ❑The only difference in 4 variants is in the implementation in commits and checkpoints.

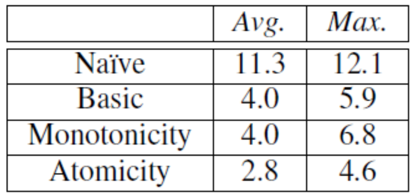
Experiment : TPC-W Benchmark

- Models online bookstore with queries asking for availability of products and an update workload that involves that placement of order:
  - retrieve the customer record from the DB
  - Search for 6 specific products
  - Place orders for 3 of the 6 products
- Purpose of the experience:
  - To study the running times and cost of the transaction for different consistency levels.
  - To study the impact of the checkpoint interval parameter on the cost.

Experiment : Running Time

❑Each transaction simulates about 12 clicks (each for 1 sec) of a user

❑Observation: The higher the level of consistency, the lower the overall running times



	Avg.	*Max.*
Naïve	11.3	12.1
Basic	4.0	5.9
Monotonicity	4.0	6.8
Atomicity	2.8	4.6

Table 3: Running Time per Transaction [secs]

❑Naïve has the highest running time because it writes all affected pages of the transaction directly to S3

❑The other approaches are faster, because clients only propagate log record to SQS instead of the whole pages.

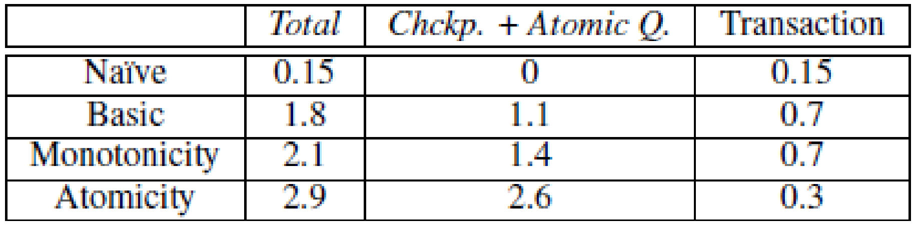
❑Improvement: latency of the commit can be reduced by sending several messages in parallel to S3 and SQS.

Experiment: Cost

❑Shows overall cost / 1000 transactions.

❑The cost was computed by running a large number of transactions, taking the cost measurements of AWS, and dividing the total cost by the number of transaction.

❑Observation: Cost increases as the level of consistency increases.



	Total	*Chckp. + Atomic Q.*	Transaction
Naïve	0.15	0	0.15
Basic	1.8	1.1	0.7
Monotonicity	2.1	1.4	0.7
Atomicity	2.9	2.6	0.3

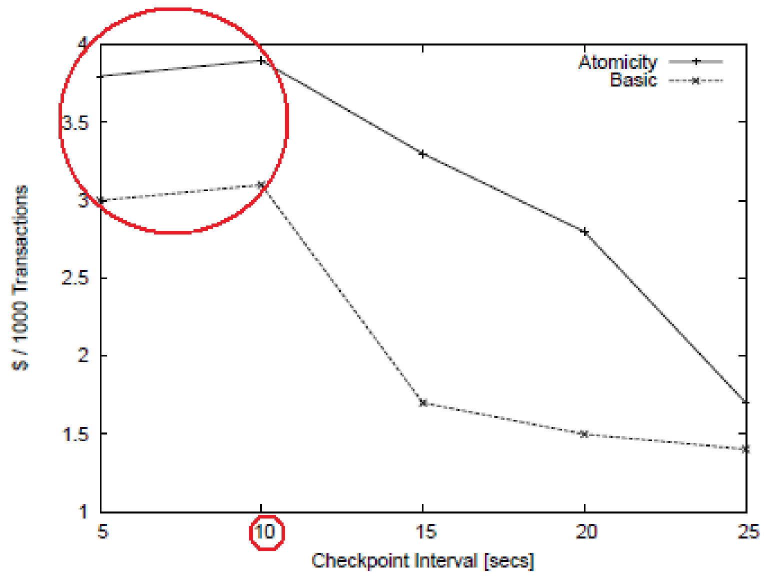
Table 4: Cost per 1000 Transactions [$]

❑Interaction with SQS is expensive.

❑Improvement: Cost can be reduced by setting the checkpoint interval to a larger value which would decrease the freshness of the data

Experience : Vary Checkpoint Interval

- Shows the cost per 1000 transaction as a function of the checkpoint internal for Basic and Atomicity approaches.
- Monotonicity approach is between Basic and Atomicity
- Observation:
  - Checkpoint interval below 10 seconds effectively involved initiating a checkpoint for every update that was committed.
  - Increasing the checkpoint interval decrease the cost. With a checkpoint interval above 10 seconds, the cost is quickly reduced in both Atomicity and Basic approaches.



Cost per 1000 Transacts., Vary Checkpoint Interval

Closing Observation

❑Utility computing is not attractive for high-performance transaction processing.

❑In this paper, strict consistency is abandon for scalability and availability, but there might be scenarios where ACID properties are more important than scalability and availability

❑The system described is not able to carry out chained I/O to scan through several pages on S3.

❑Right security infrastructure is needed for S3 system

Question?