Transactional Consistency and Automatic Management in an Application Data Cache

Dan Ports
Austin Clements Irene Zhang
Samuel Madden Barbara Liskov

MIT CSAIL

Modern web applications face immense scaling challenges

increasingly complex, personalized content

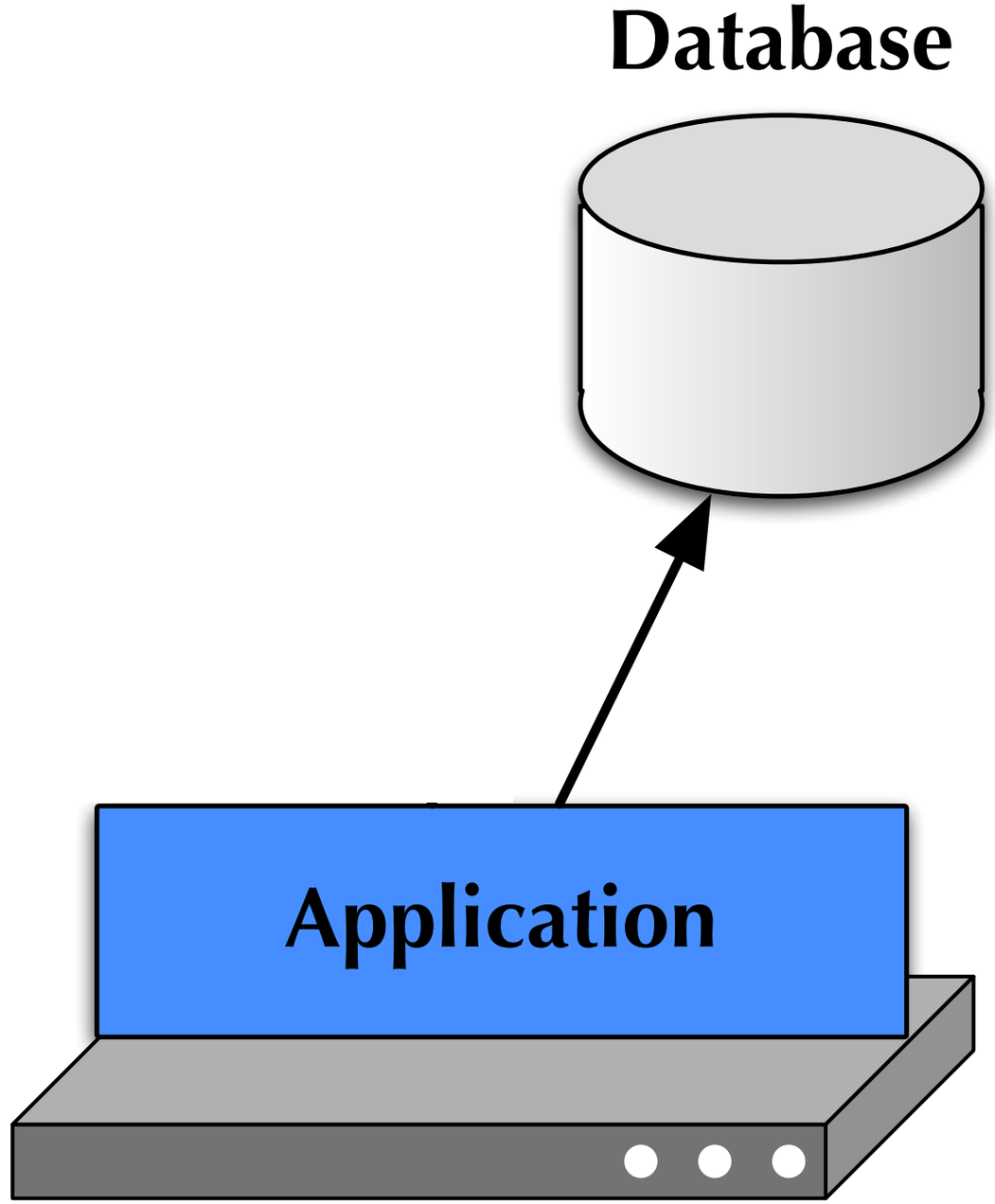
e.g. Facebook, MediaWiki, LiveJournal...

Existing caching techniques are less useful

whole-page caches: foiled by personalization

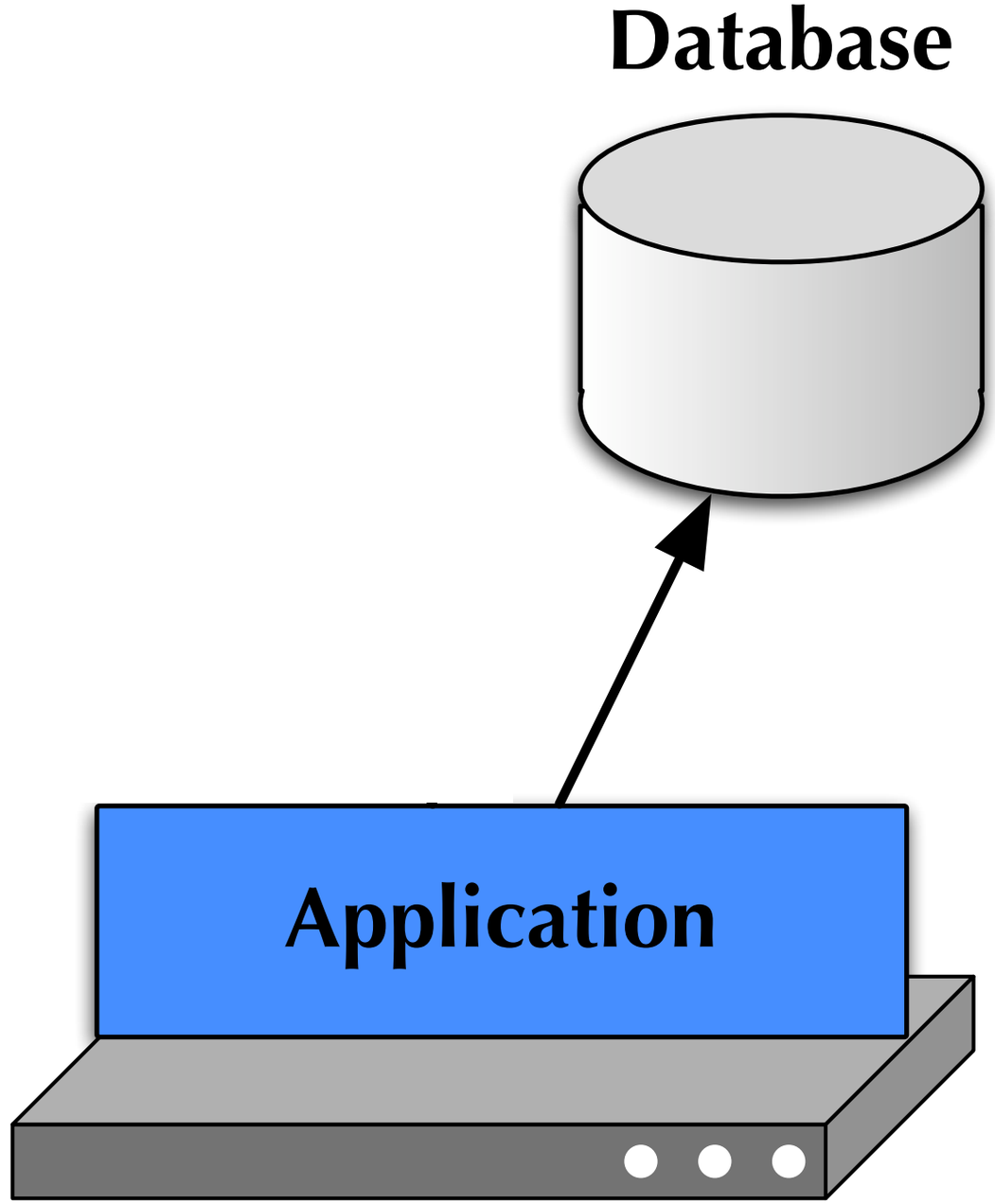
database caches: more processing is being done in the *application* layer

Application-Level Caching



Application-Level Caching

e.g. memcached,
Java object caches

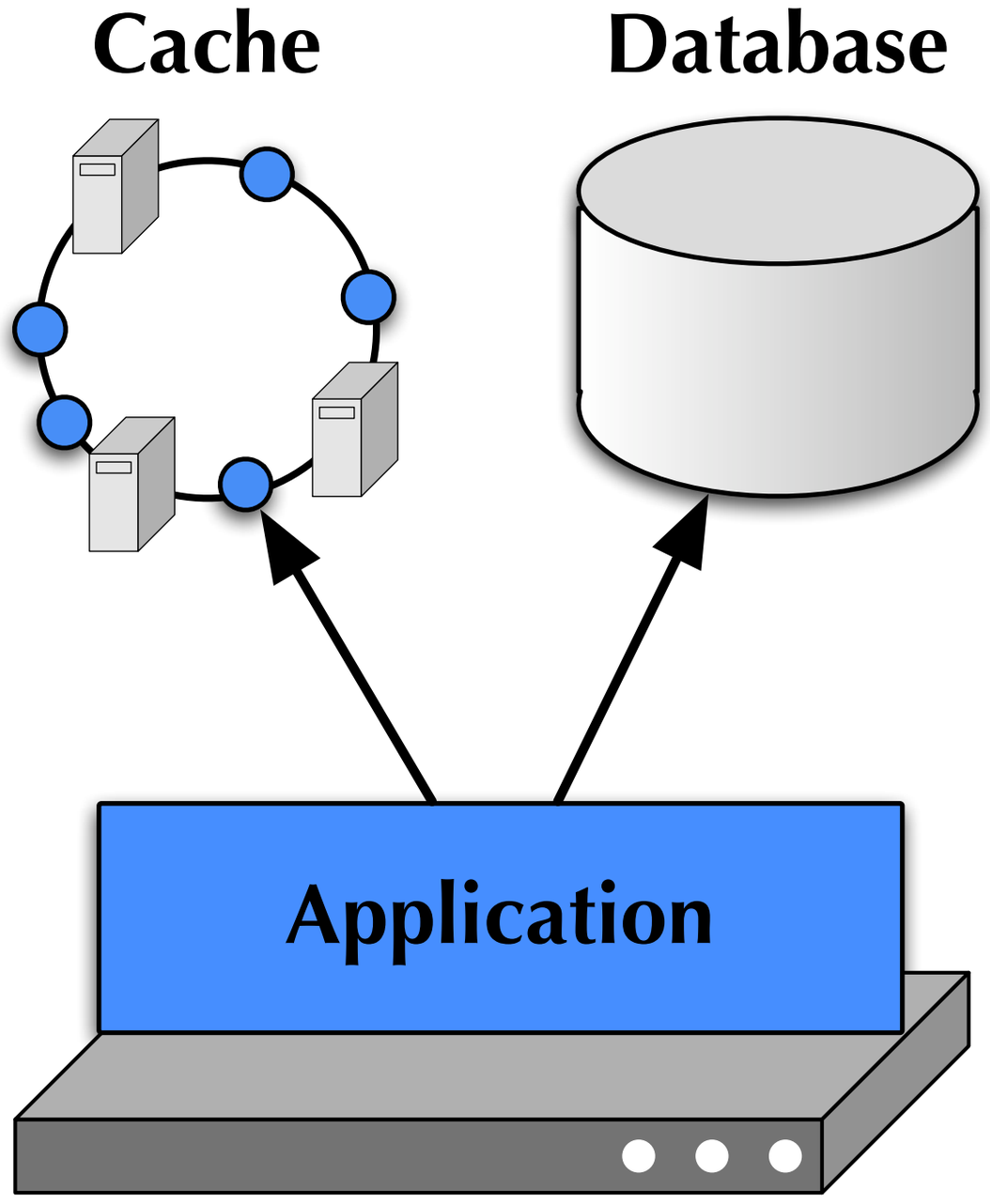


Application-Level Caching

e.g. memcached,
Java object caches

very lightweight
in-memory caches

stores ***application*** objects
(computations),
i.e.:
not a database replica
not a query cache



Why Cache Application Data?

Cache higher-level data closer to app needs:
DB queries, complex structures, HTML fragments

Can separate common and customized content

Reduces database load

Reduces application server load

- this matters too (application servers aren't cheap!)

Existing Caches Add To Application Complexity

No transactional consistency

- violates guarantees of the underlying DB
- app. code must deal with transient anomalies

Hash table interface leaves apps responsible for:

- naming and retrieving cache entries
- keeping cache up-to-date (invalidations)

Harder Than You Think!

Naming: cache key must uniquely identify value

- MediaWiki stored list of recent changes with same key regardless of # days requested (#7541)

Invalidations: require reasoning globally about entire application

- After editing wiki page, what to invalidate?

Harder Than You Think!

Naming: cache key must uniquely identify value

- MediaWiki stored list of recent changes with same key regardless of # days requested (#7541)

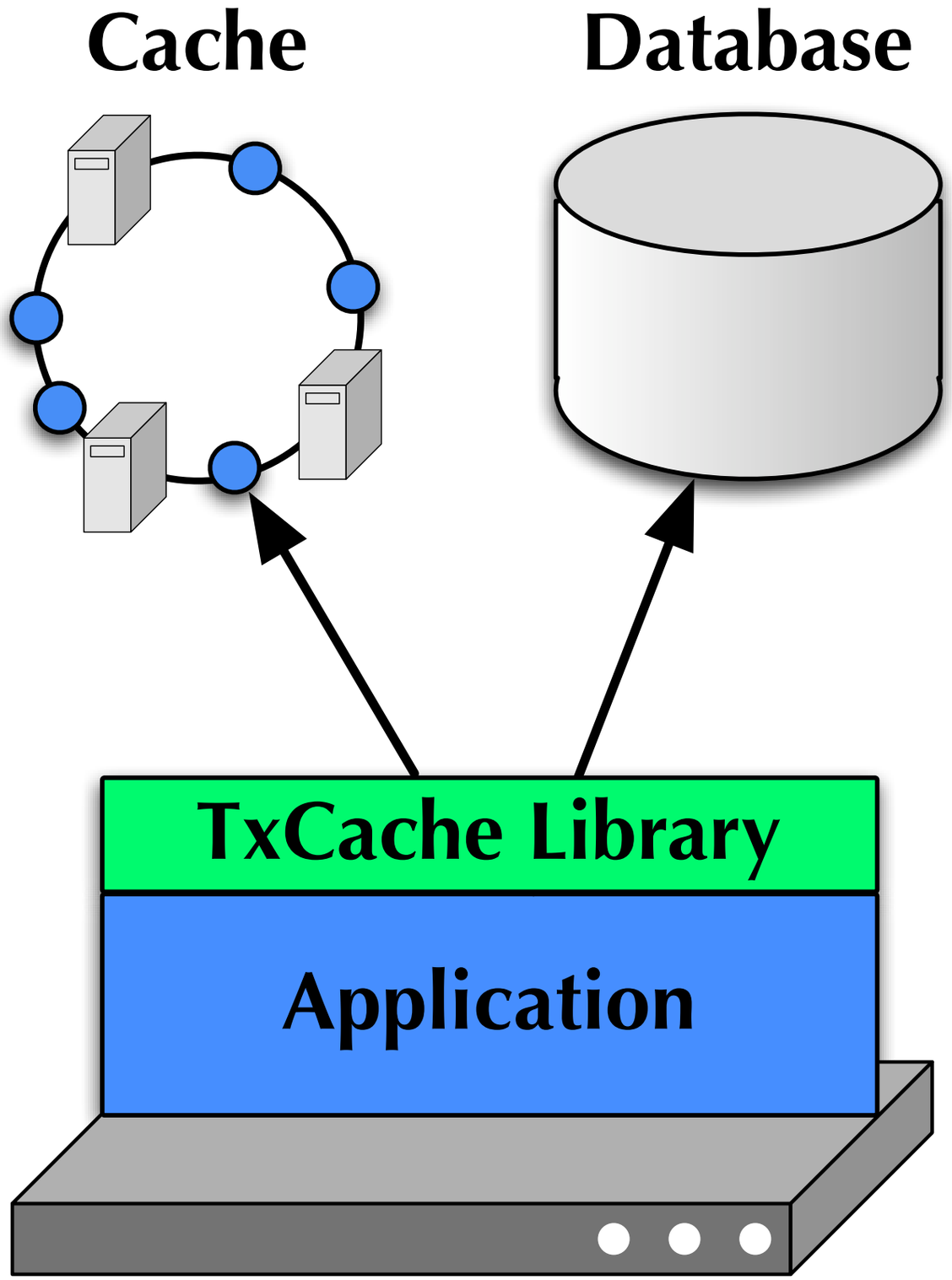
Invalidations: require reasoning globally about entire application

- After editing wiki page, what to invalidate?
- Forgot editor's *User* object – contains edit count (#8391)

Introducing TxCache

Our cache provides:

- **transactional consistency:** serializable, point-in-time view of data, whether from cache or DB
- **bounded staleness:** improves hit rate for applications that accept old (but consistent) data
- **simpler interface:**
  applications mark functions cacheable;
  TxCache caches their results,
  including naming and invalidations



- TxCache library hides complexity of cache management
- Integrates with new cache server, minor DB modifications (Postgres; <2K lines changed)
- Together, ensure whole-system transactional consistency

TxCache Interface

- beginRO(*staleness*), commit(),
  beginRW(), abort()
- make-cacheable(*fn*)
  where *fn* is a side-effect-free function that depends only on its arguments and the database state
  - ➔ *fn* returns cached result of previous call with same inputs if still consistent w/ DB

TxCache Interface

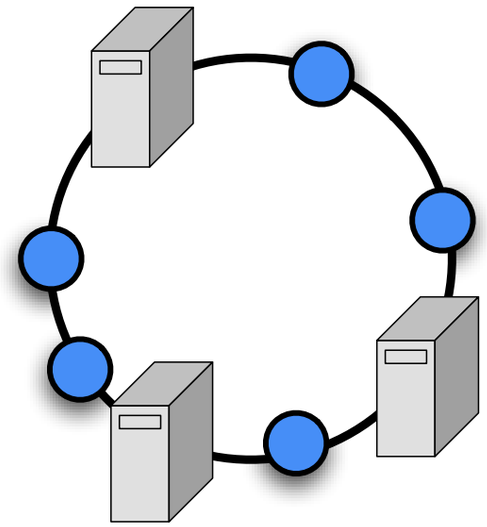
- beginRO(*staleness*), commit(),
  beginRW(), abort()
- make-cacheable(*fn*)
  where *fn* is a side-effect-free function that depends only on its arguments and the database state
  ➔ *fn* returns cached result of previous call with same inputs if still consistent w/ DB

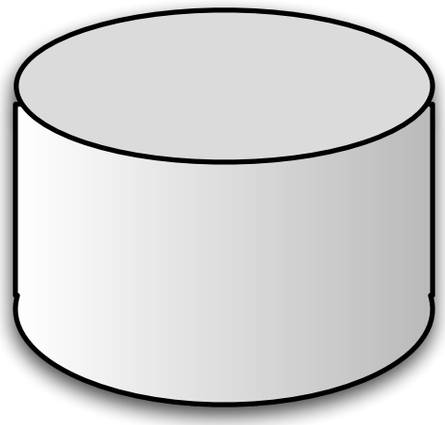
That's it.

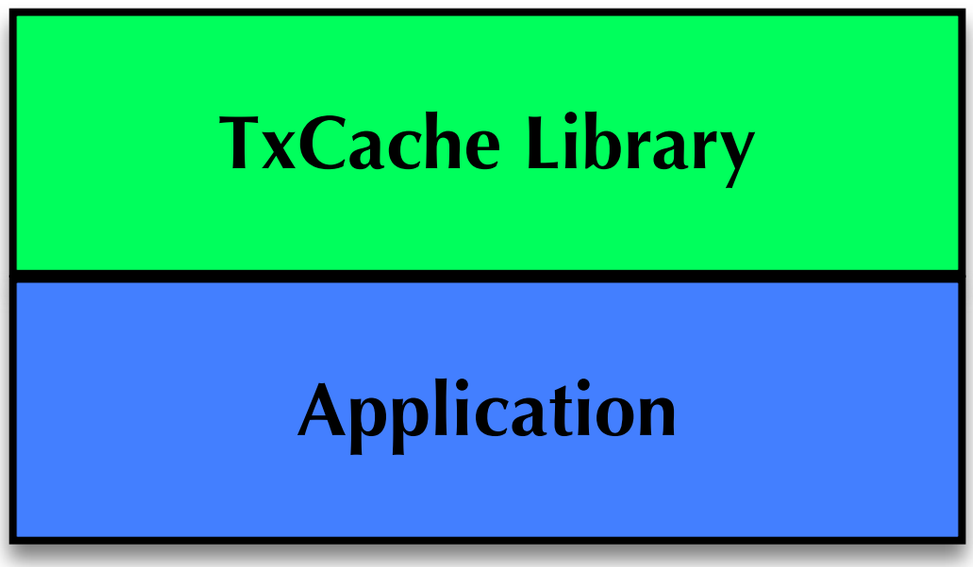
TxCache Interface

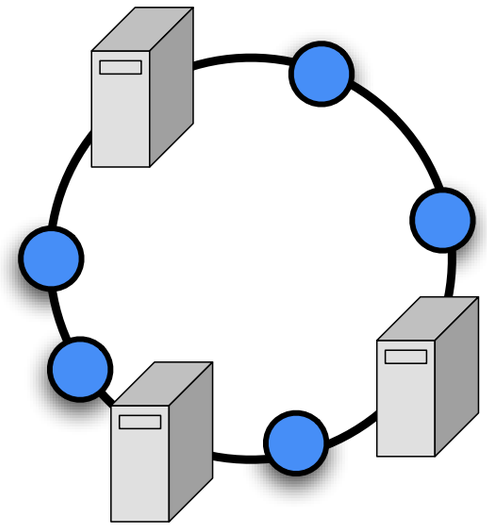
- beginRO(*staleness*), commit(),
  beginRW(), abort()
- make-cacheable(*fn*)
  where *fn* is a side-effect-free function that depends only on its arguments and the database state
  ➔ *fn* returns cached result of previous call with same inputs if still consistent w/ DB

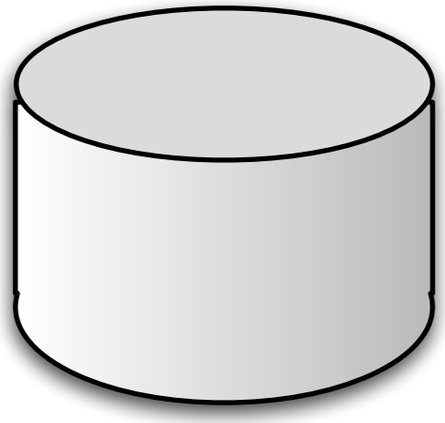
That's it.
Really!

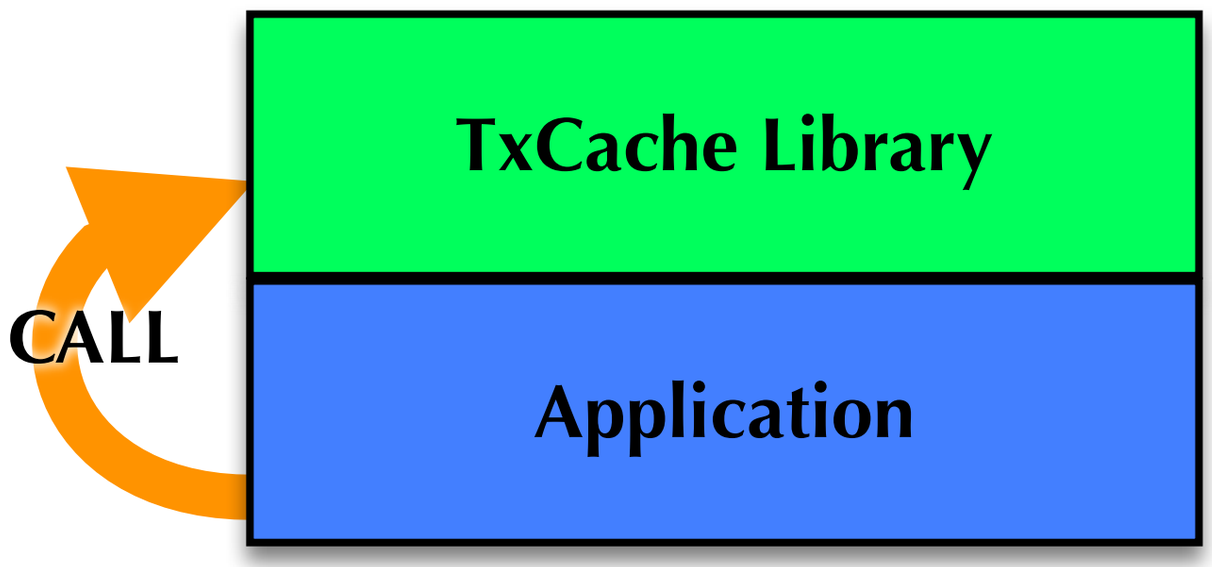


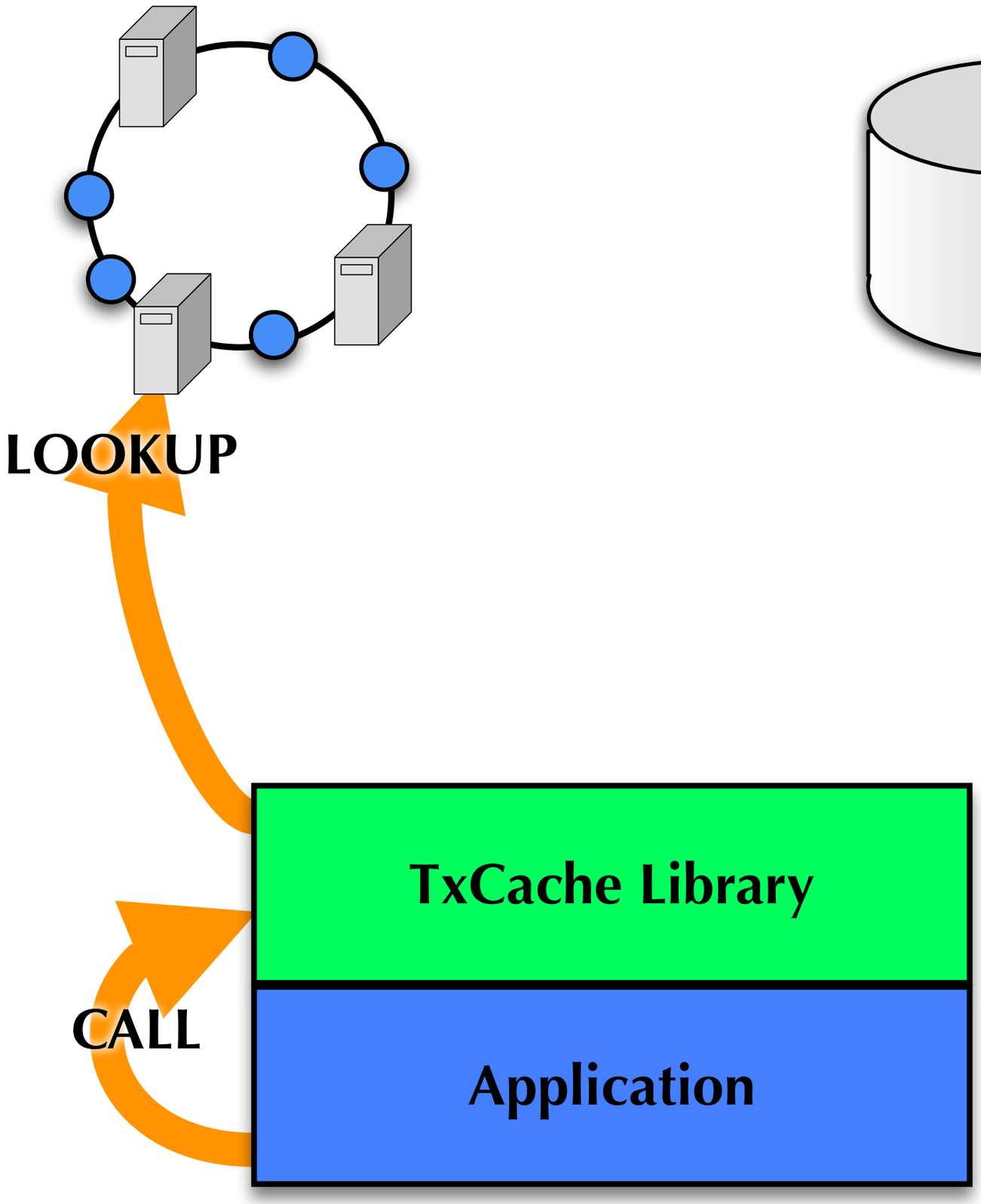


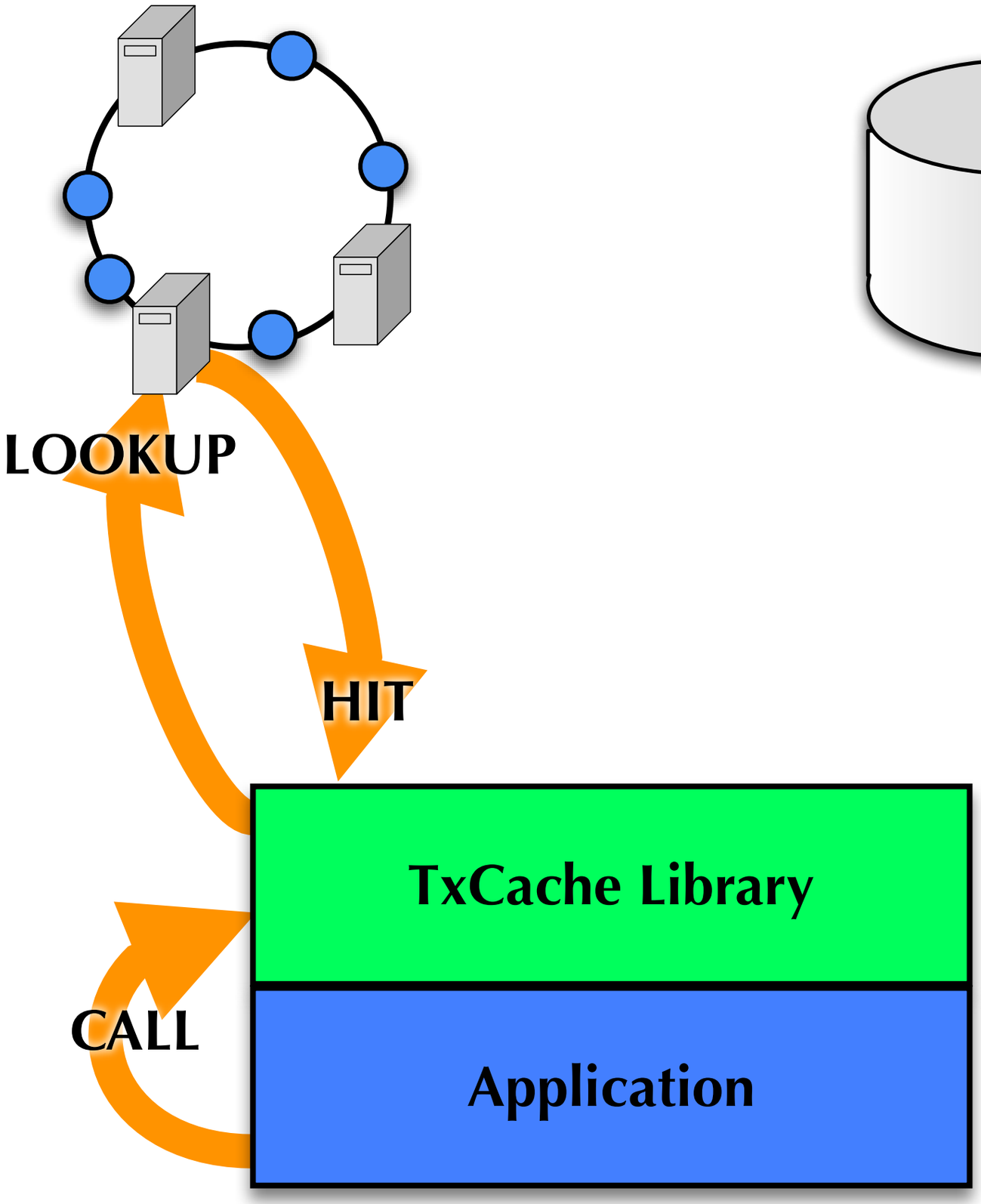

TxCache Library
Application

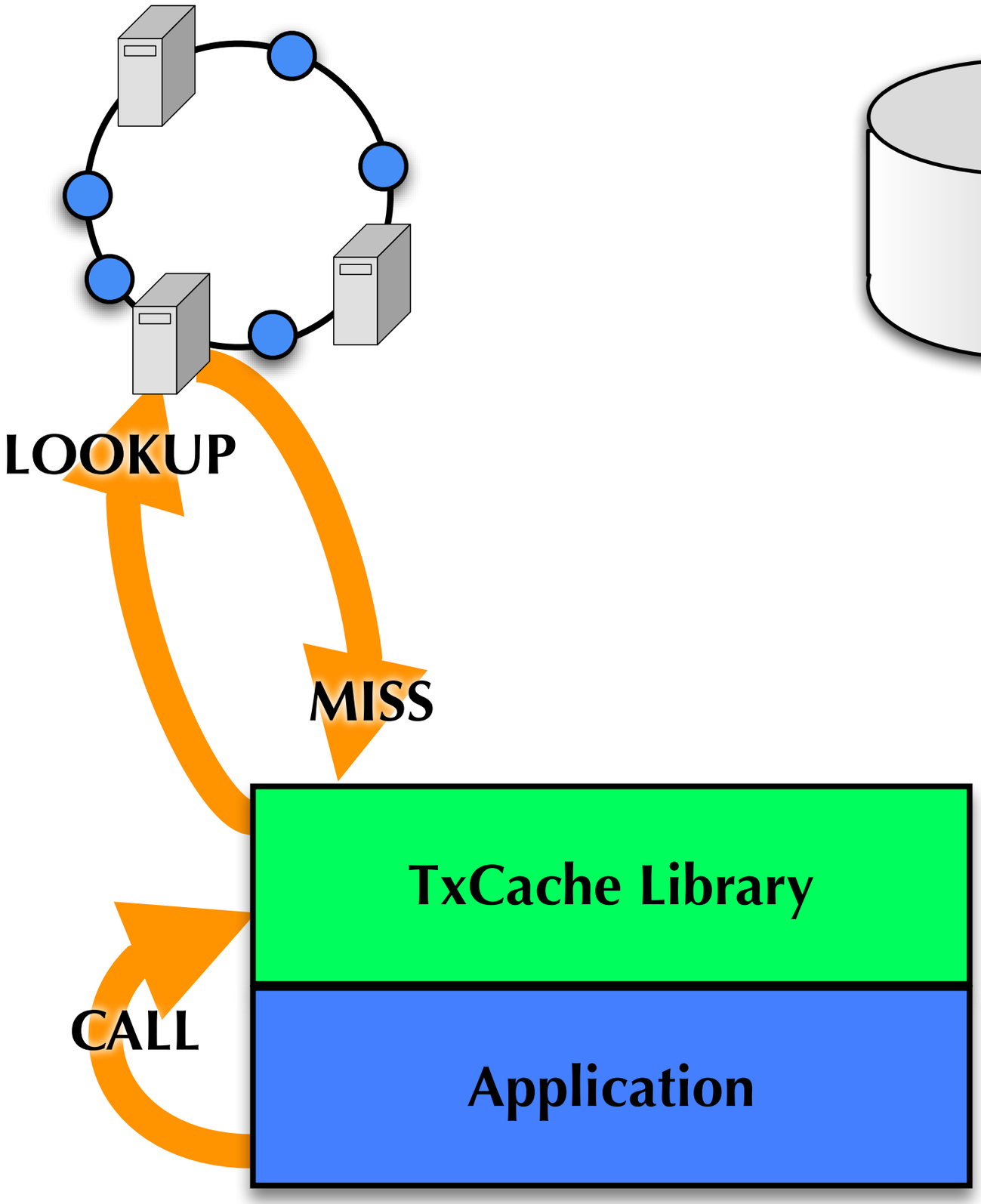


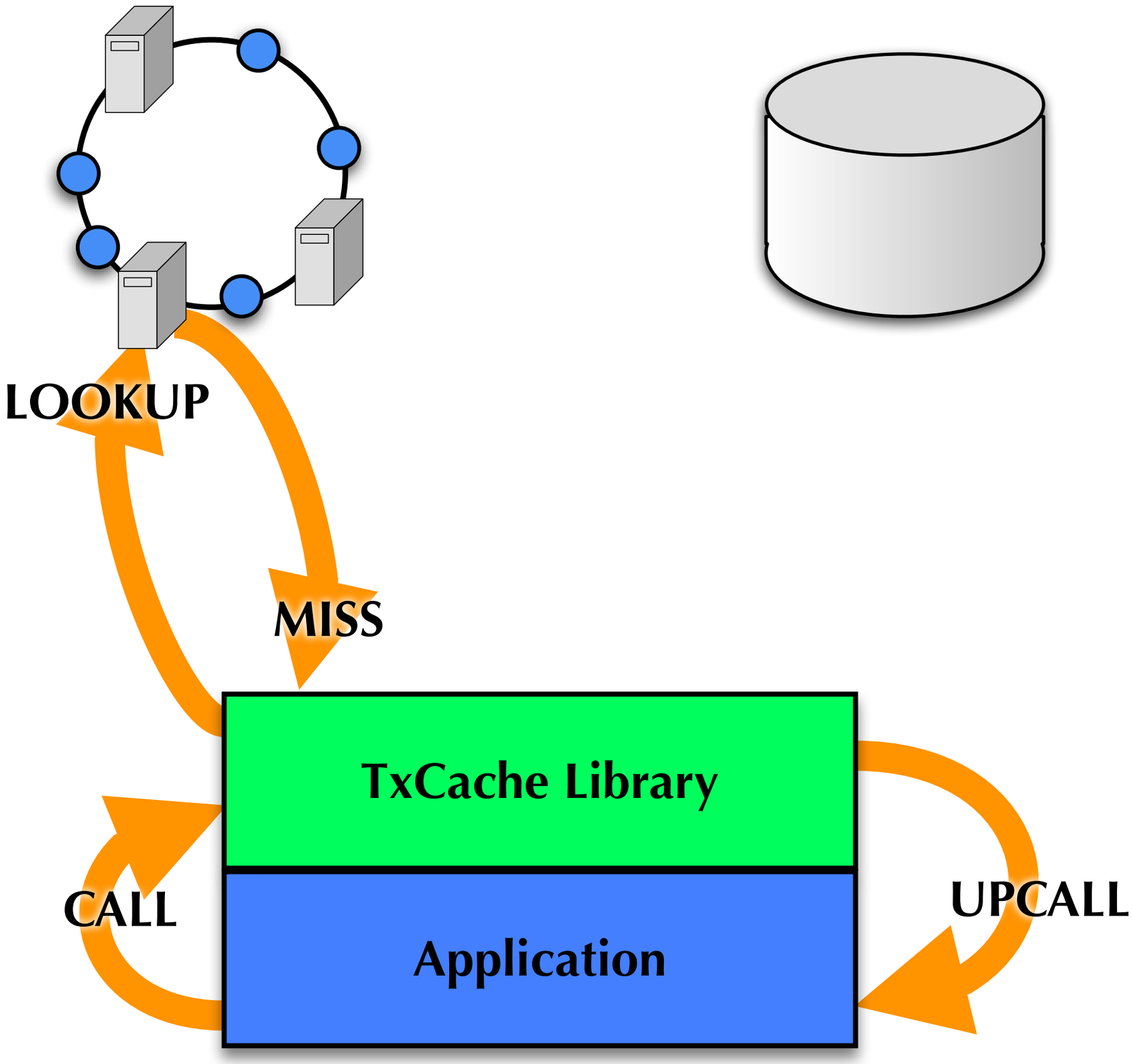


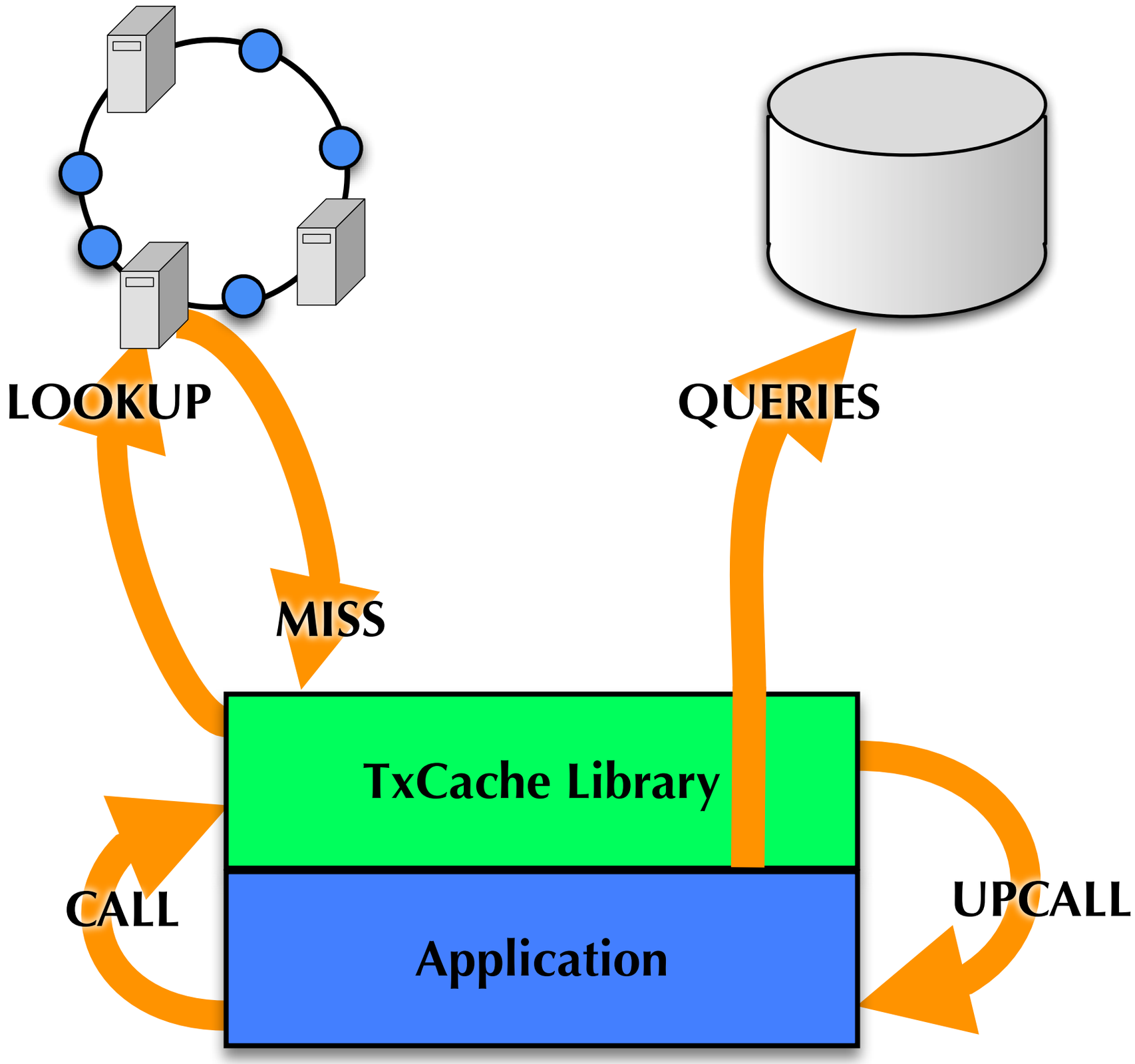

TxCache Library
Application
CALL

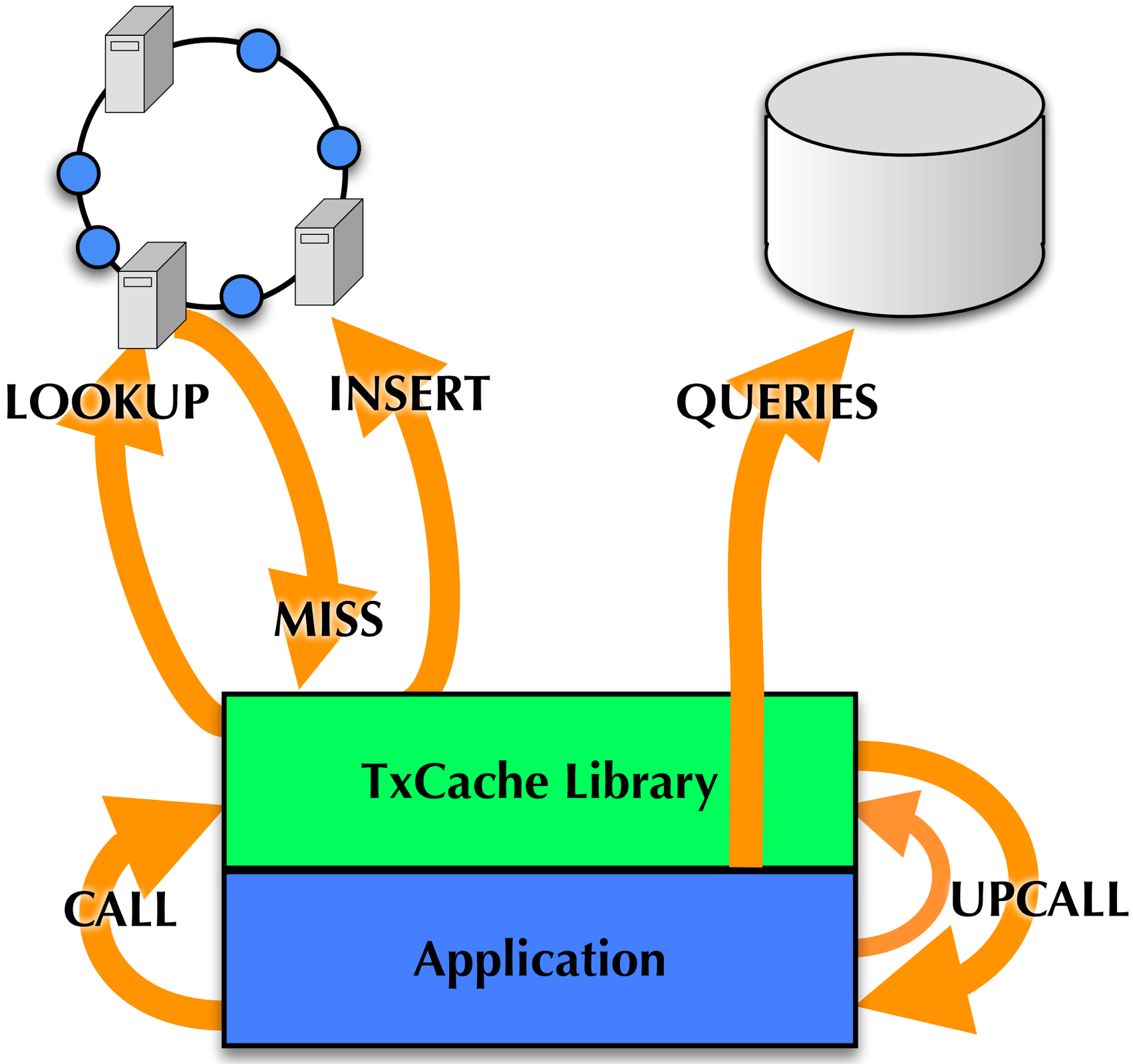

LOOKUP
TxCache Library
CALL
Application

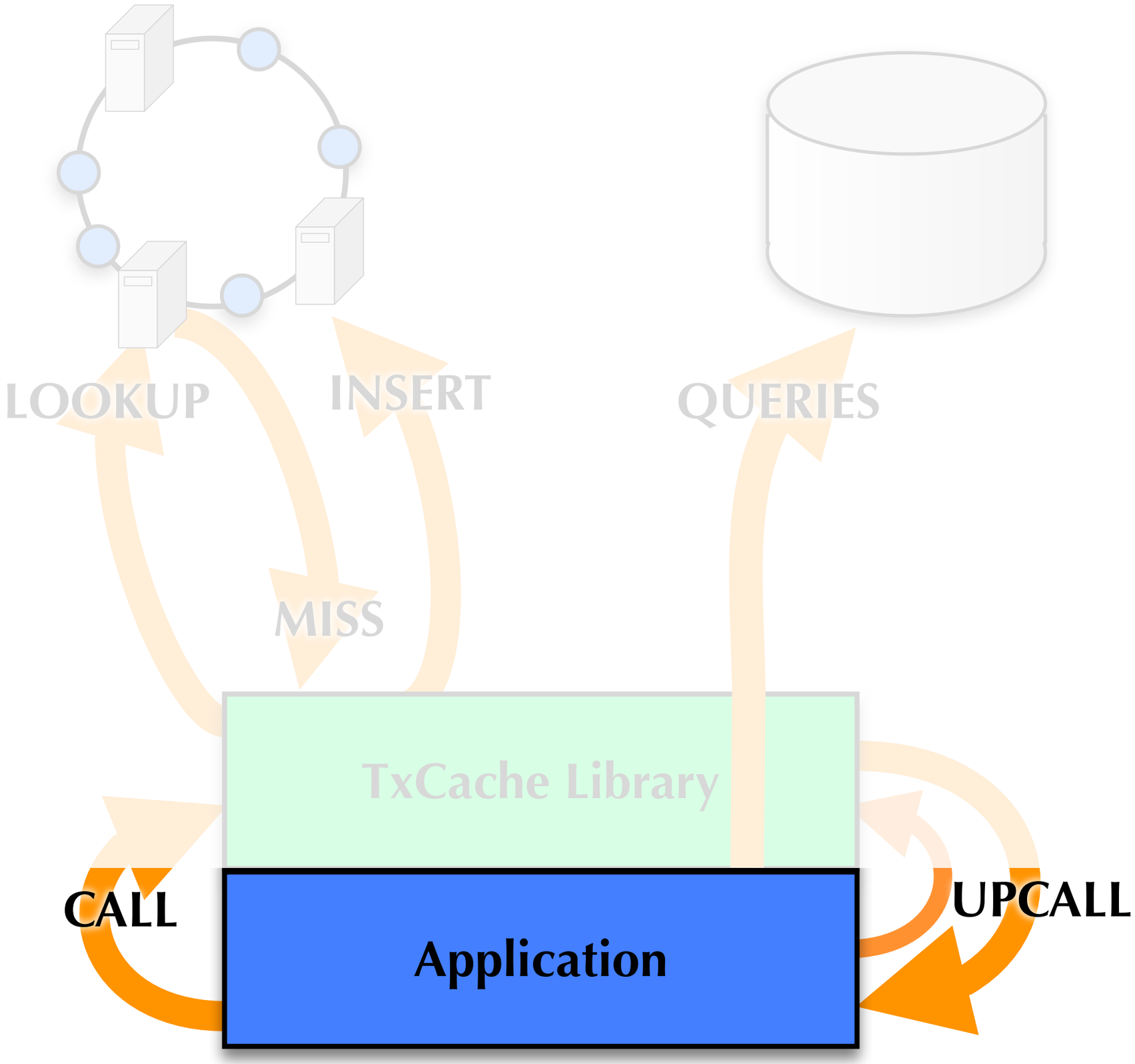

LOOKUP
HIT
CALL
TxCache Library
Application


LOOKUP
MISS
CALL
TxCache Library
Application


LOOKUP
MISS
TxCache Library
Application
CALL
UPCALL


LOOKUP
MISS
QUERIES
TxCache Library
Application
CALL
UPCALL


LOOKUP
INSERT
QUERIES
MISS
TxCache Library
Application
CALL
UPCALL


LOOKUP
INSERT
QUERIES
MISS
TxCache Library
CALL
Application
UPCALL

Outline

1. Application-Level Caching
2. TxCache Interface
3. **Ensuring Transactional Consistency**
4. Automating Invalidations
5. Evaluation

Consistency Approach

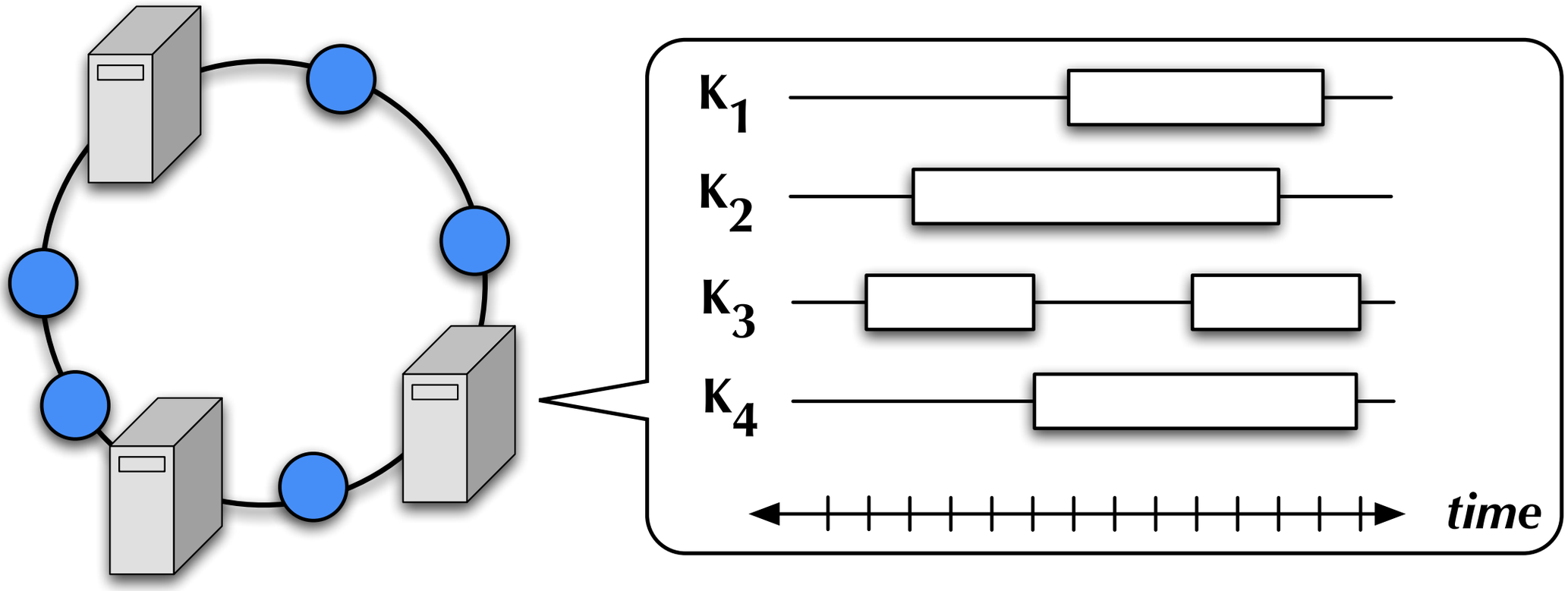
Goal: all data seen in a transaction reflects single point-in-time snapshot

- Assign timestamp to transaction
- Know the ***validity interval*** of each object in cache or database:
  set of timestamps when it was valid
- Then: transaction can read data if data's validity interval contains txn's timestamp

A Versioned Cache

Cache entries tagged with validity intervals

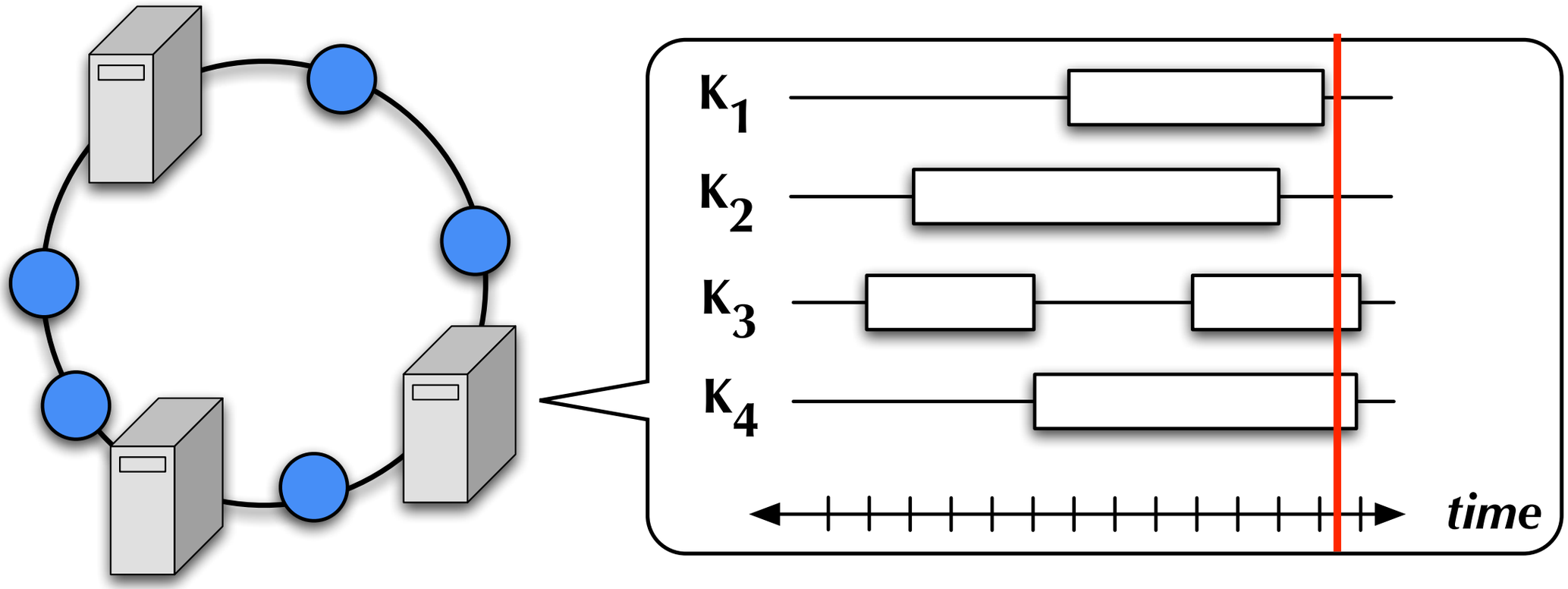
- each entry one immutable version of an object
- allows lookup for value valid at certain time



A Versioned Cache

Cache entries tagged with validity intervals

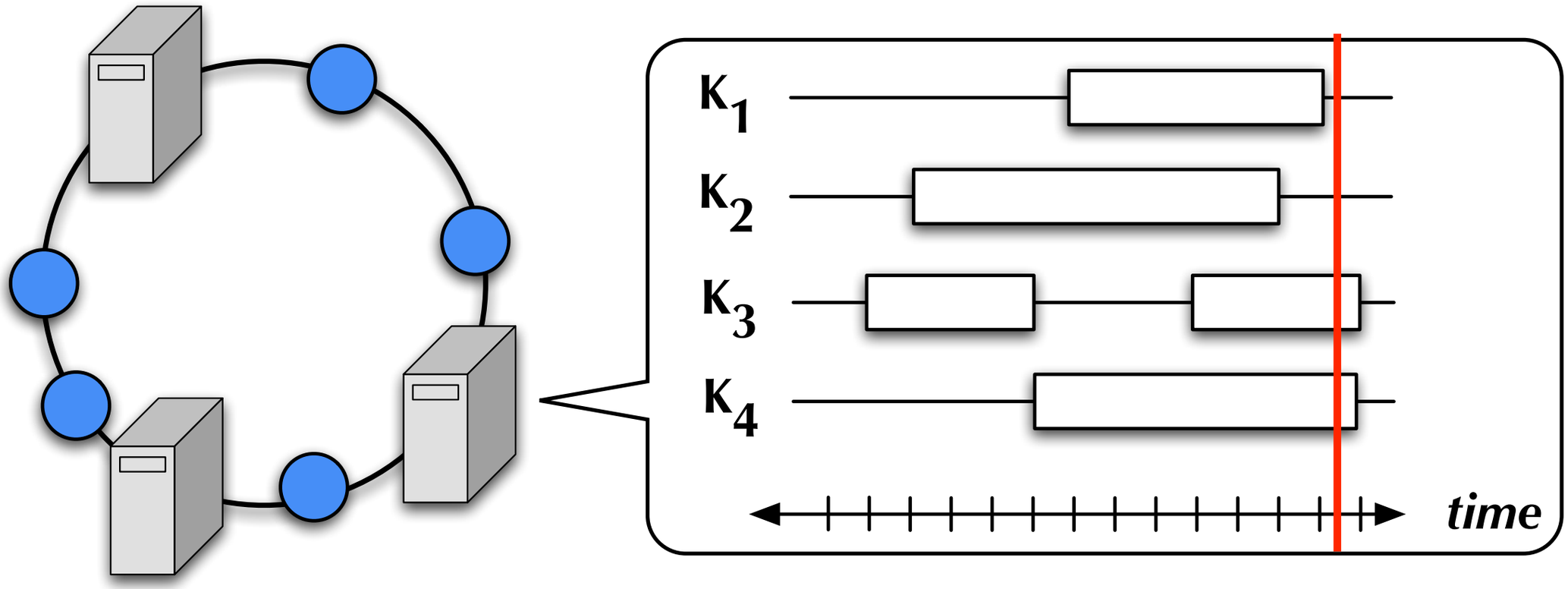
- each entry one immutable version of an object
- allows lookup for value valid at certain time



Staleness

Assign transaction an earlier timestamp

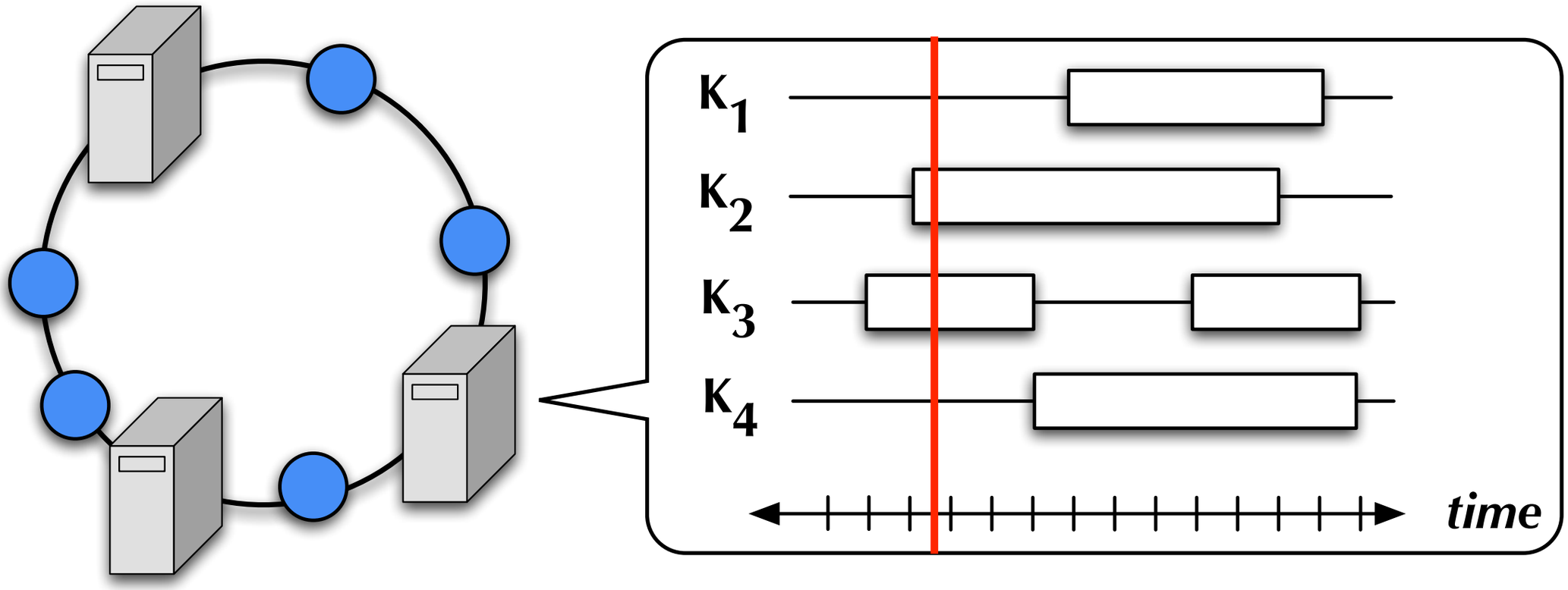
- if consistent with application requirements
- allows cached data to be used longer



Staleness

Assign transaction an earlier timestamp

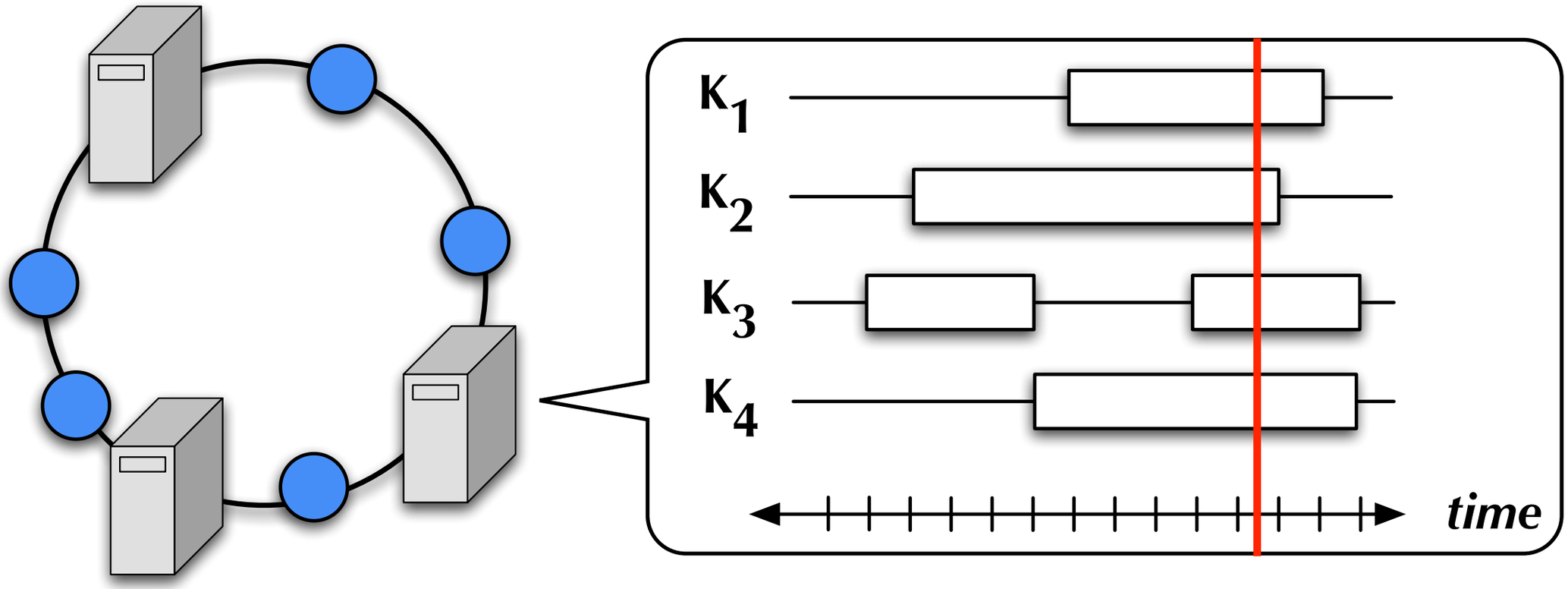
- if consistent with application requirements
- allows cached data to be used longer



Staleness

Assign transaction an earlier timestamp

- if consistent with application requirements
- allows cached data to be used longer



Staleness

Assign transaction an earlier timestamp

- if consistent with application requirements
- allows cached data to be used longer

Requires starting a DB transaction at same timestamp

- internally, snapshot isolation supports this
- added interface to expose this to cache library

Where Do Validity Intervals Come From?

Where Do Validity Intervals Come From?

Validity of an application object
= validity of the DB queries used to generate it

- library tracks query dependencies

Where Do Validity Intervals Come From?

Validity of an application object
= validity of the DB queries used to generate it

- library tracks query dependencies

Validity of a DB query
= validity of the tuples accessed to compute it

- we modify the DB to report this

Where Do Validity Intervals Come From?

Validity of an application object
= validity of the DB queries used to generate it

- library tracks query dependencies

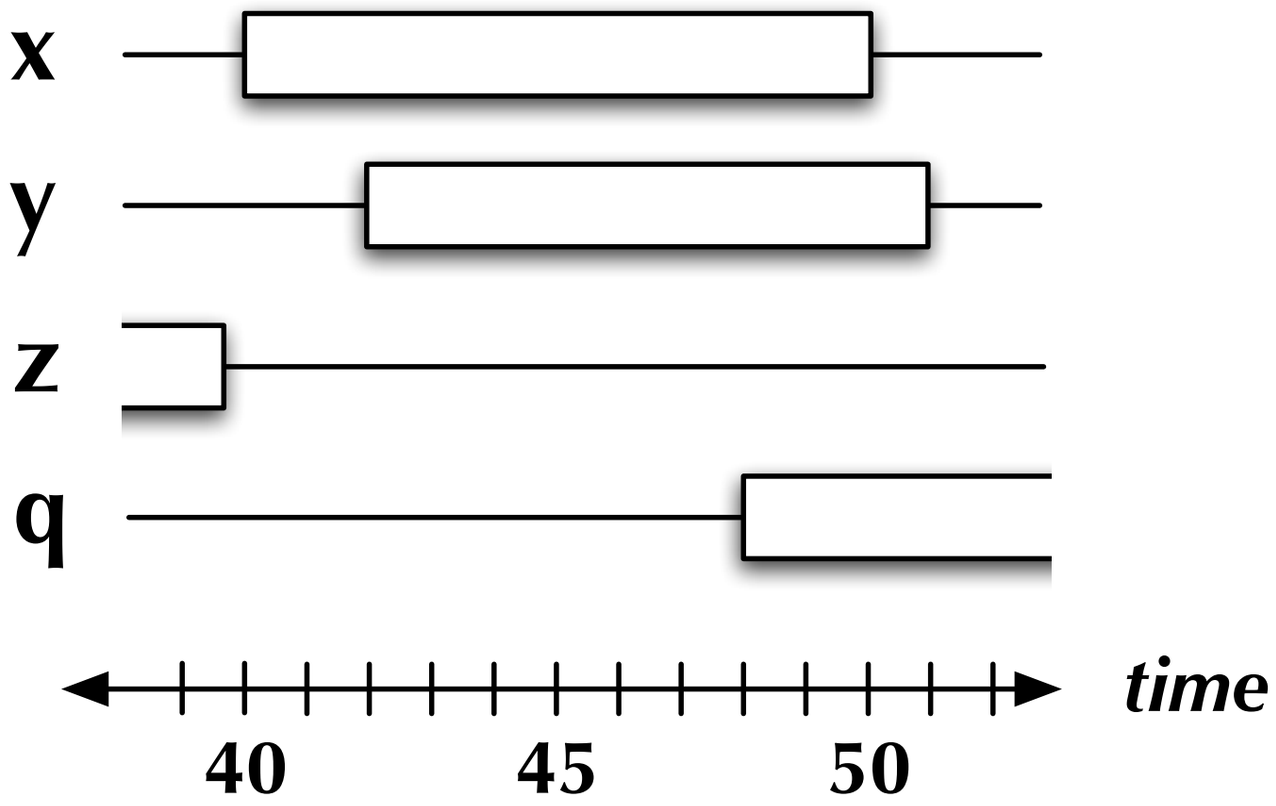
Validity of a DB query
= validity of the tuples accessed to compute it

- we modify the DB to report this

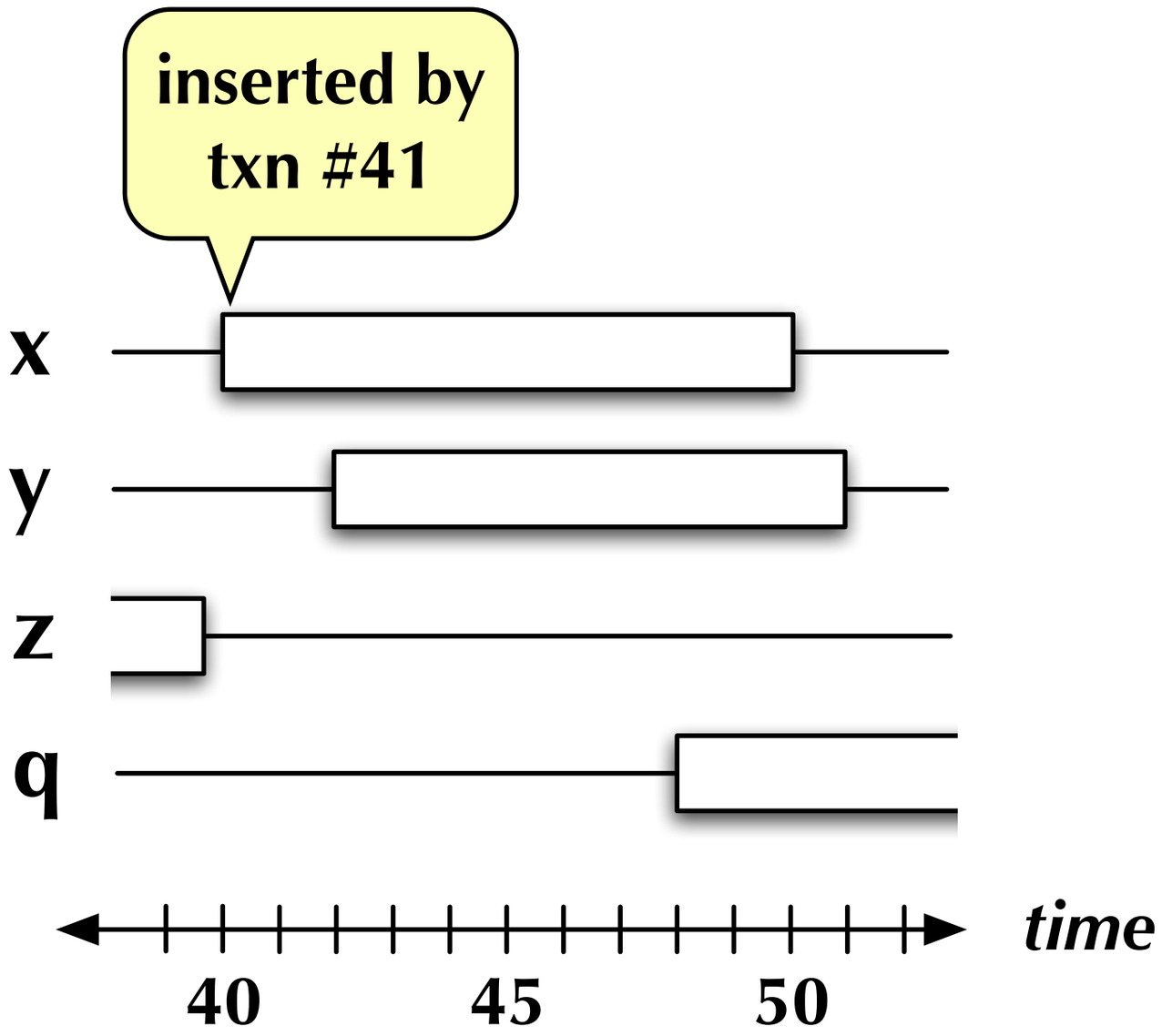
Validity of a tuple
= timestamps of creating, deleting transactions

- multiversion DBs already track this

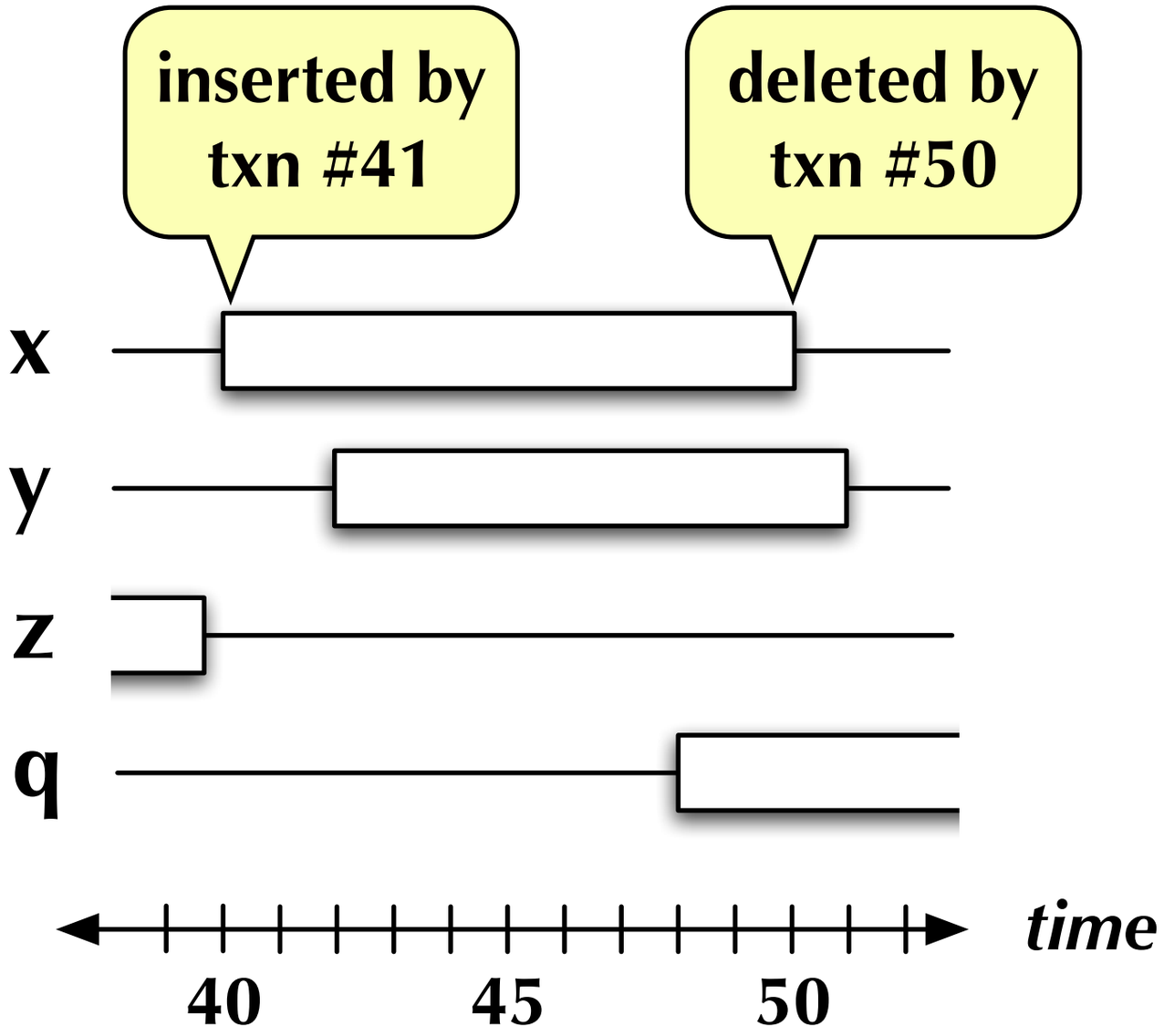
Computing Query Validity



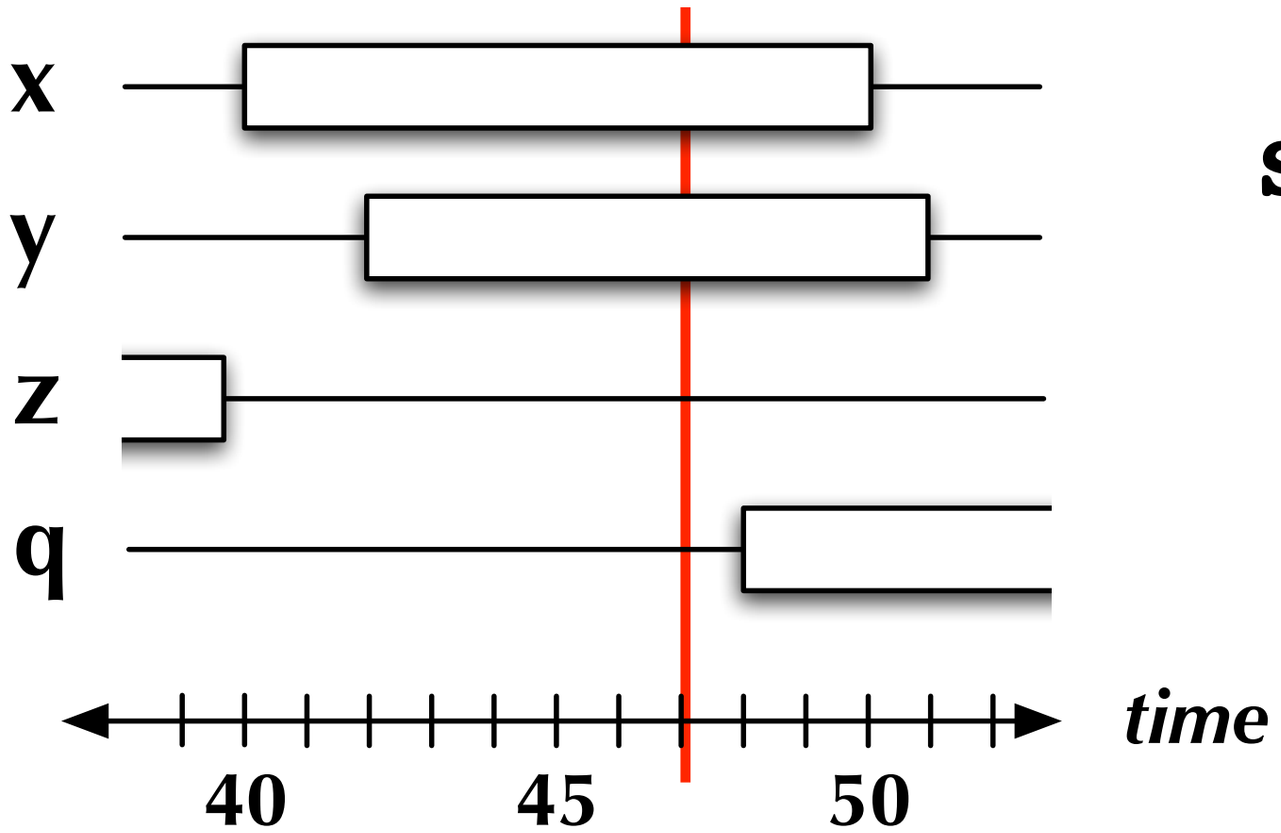
Computing Query Validity



Computing Query Validity

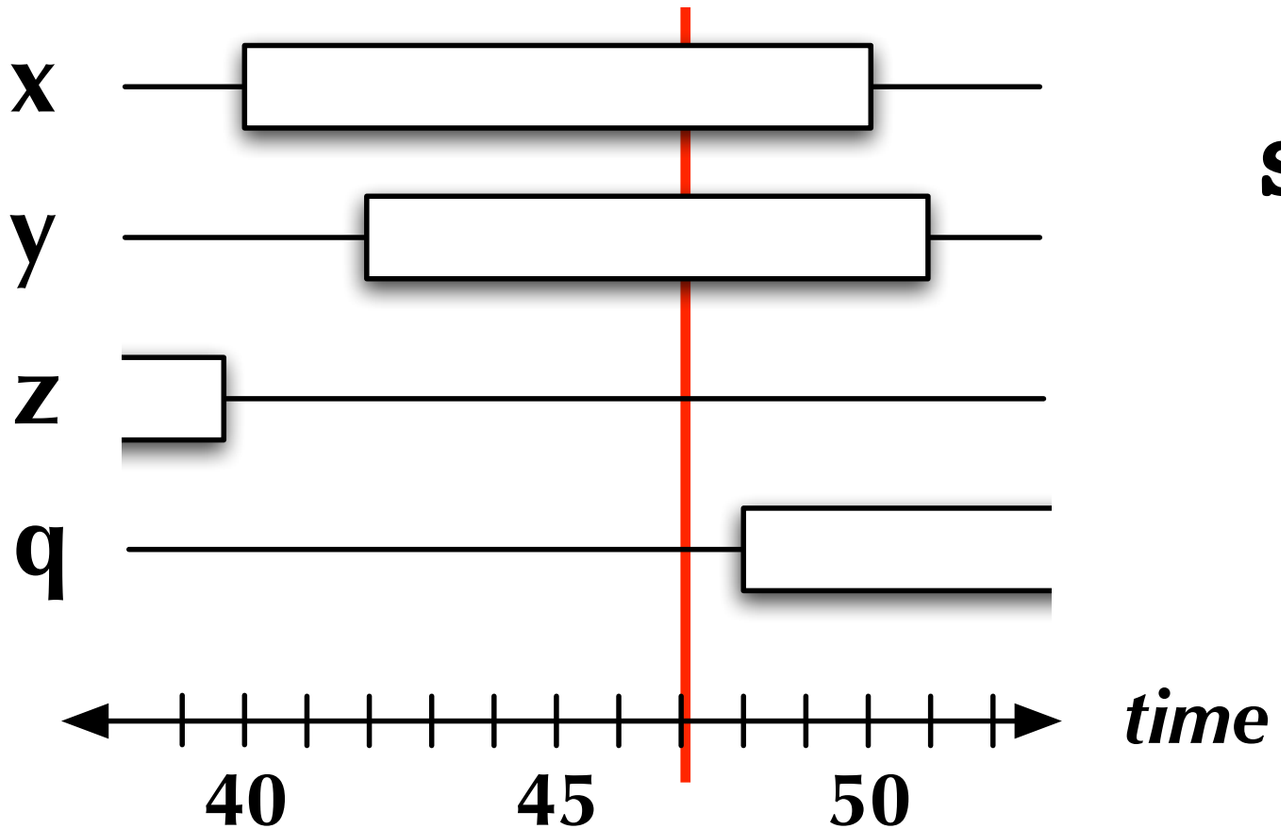


Computing Query Validity




```
SELECT * FROM ...;
```

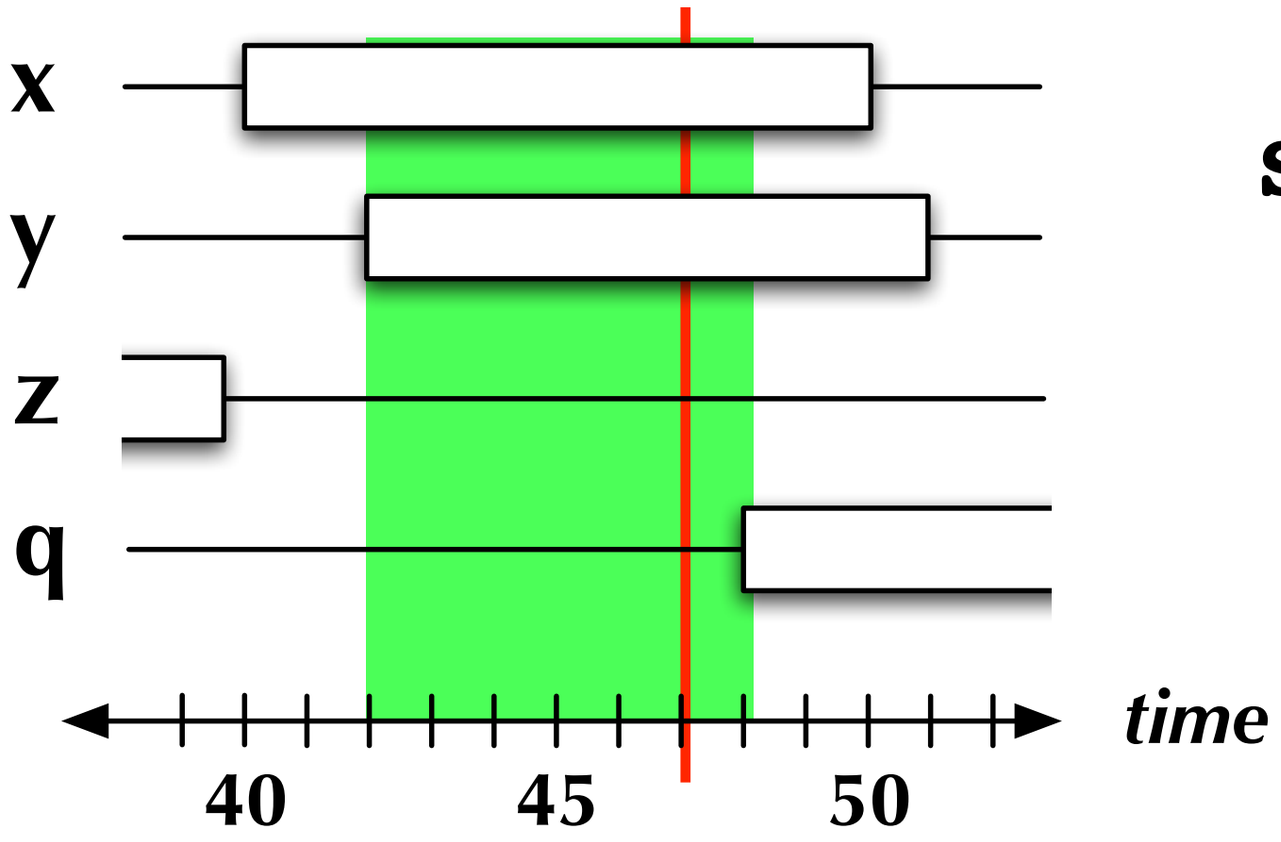
Computing Query Validity




```
SELECT * FROM ...;
  result = {x, y}
```

Computing Query Validity

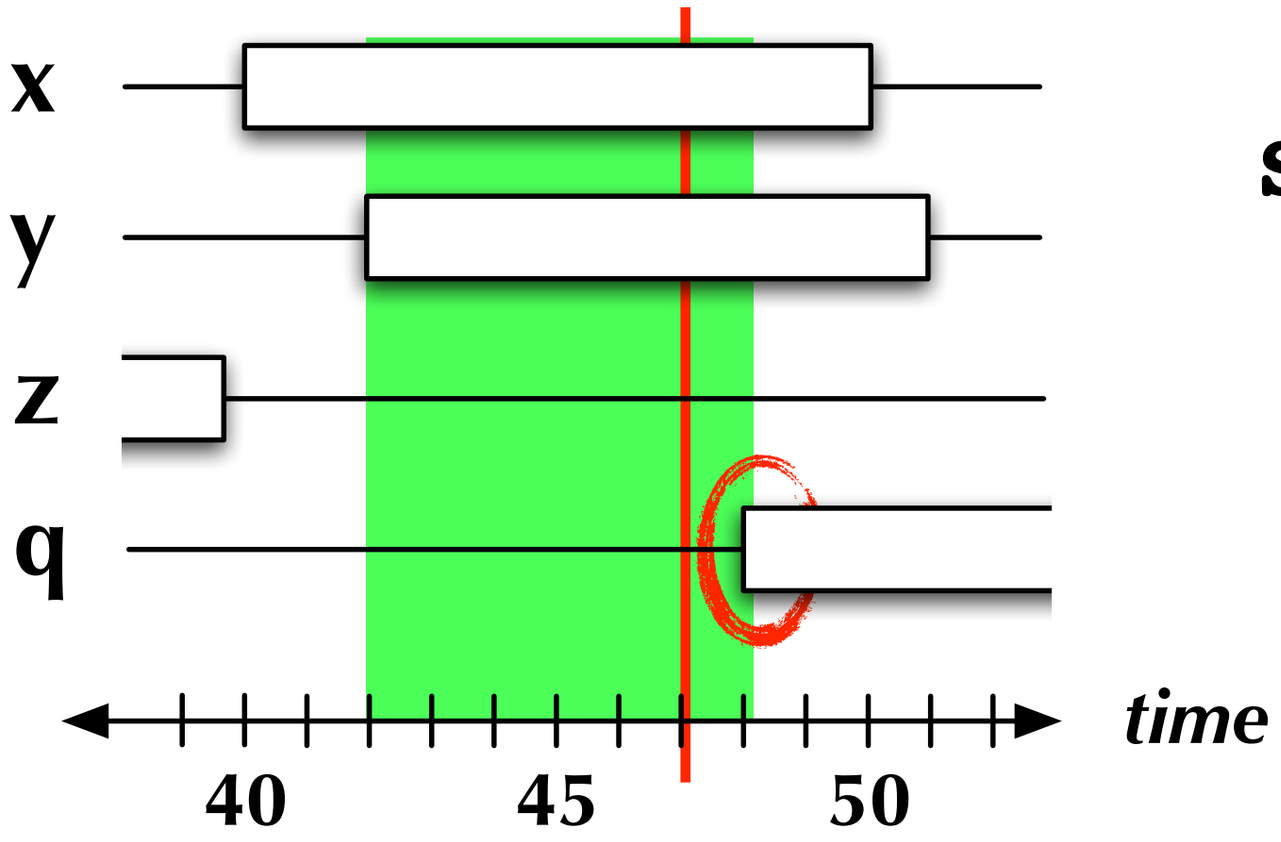
Intersect validity intervals of tuples accessed




```
SELECT * FROM ...;
  result = {x, y}
  VALIDITY [41, 48)
```

Computing Query Validity

Intersect validity intervals of tuples accessed

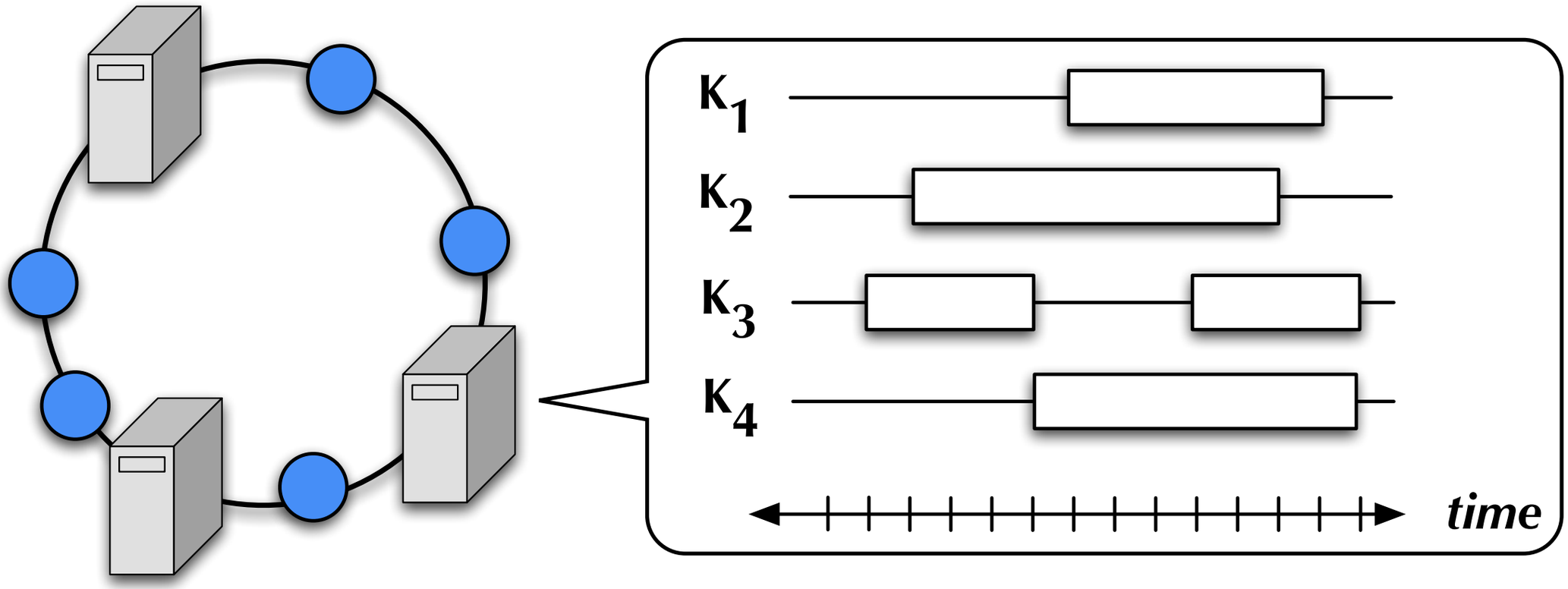



```
SELECT * FROM ...;
  result = {x, y}
  VALIDITY [41, 48)
```

Lazy Timestamp Selection

Hard to choose timestamp *a priori*

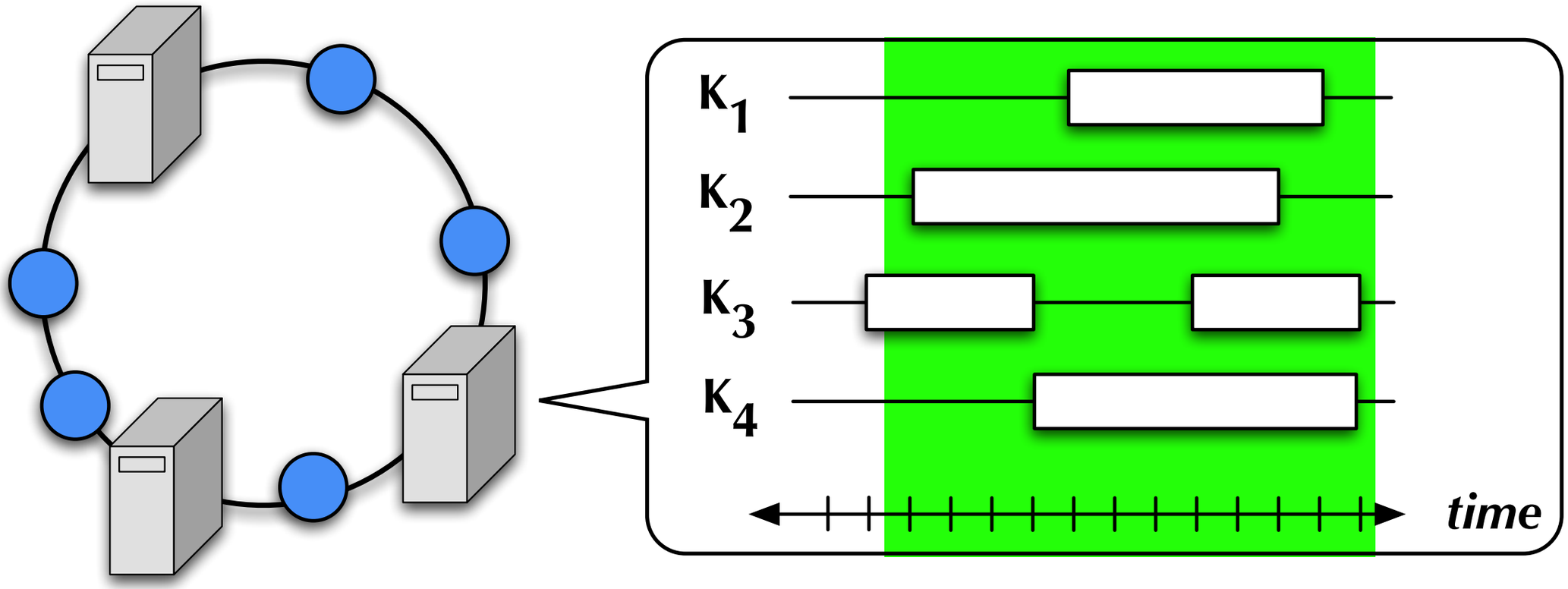
- Don't know access pattern *or* cache contents
- **Insight: don't have to choose right away!**



Lazy Timestamp Selection

Hard to choose timestamp *a priori*

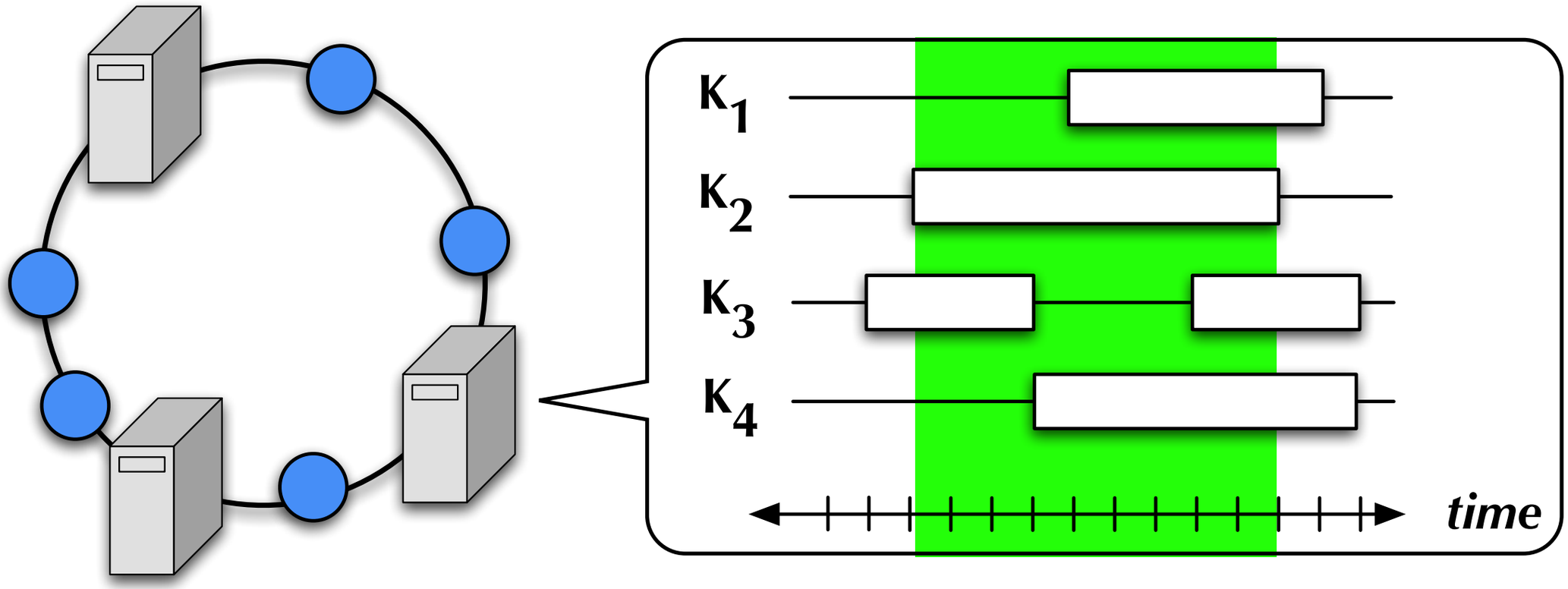
- Don't know access pattern *or* cache contents
- **Insight: don't have to choose right away!**



Lazy Timestamp Selection

Hard to choose timestamp *a priori*

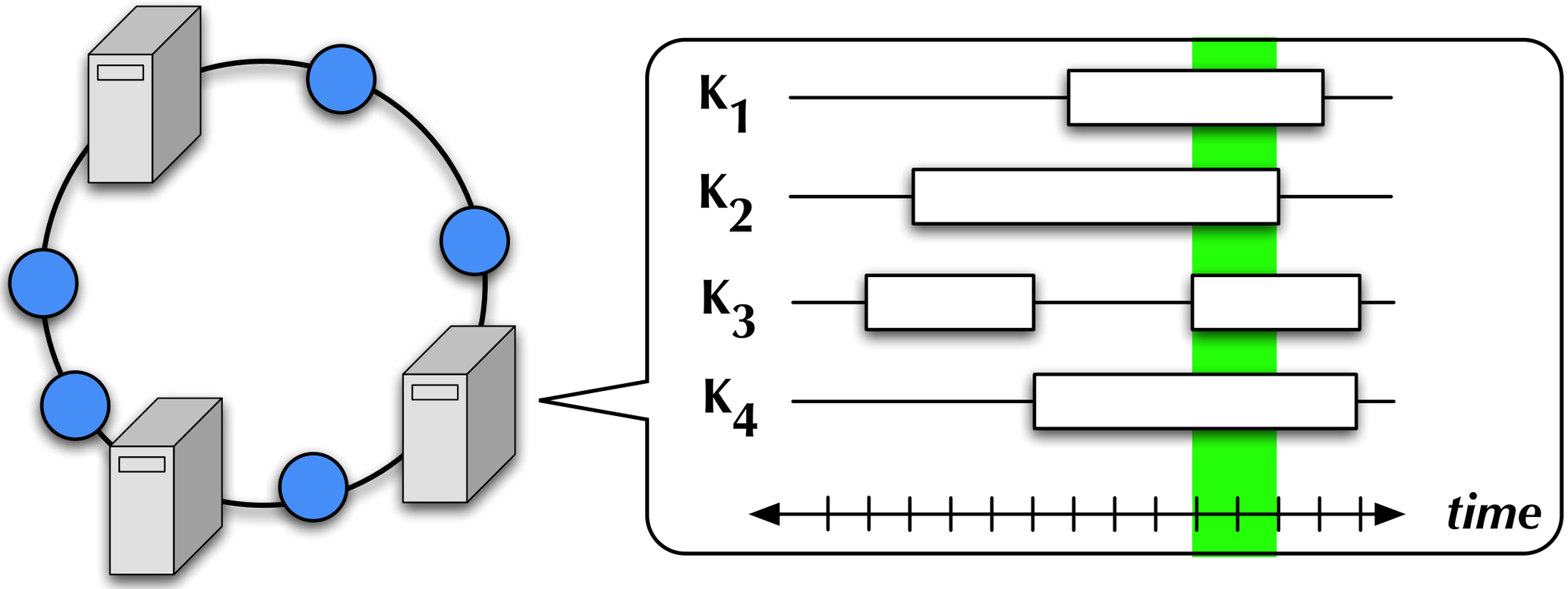
- Don't know access pattern *or* cache contents
- **Insight: don't have to choose right away!**



Lazy Timestamp Selection

Hard to choose timestamp *a priori*

- Don't know access pattern *or* cache contents
- **Insight: don't have to choose right away!**



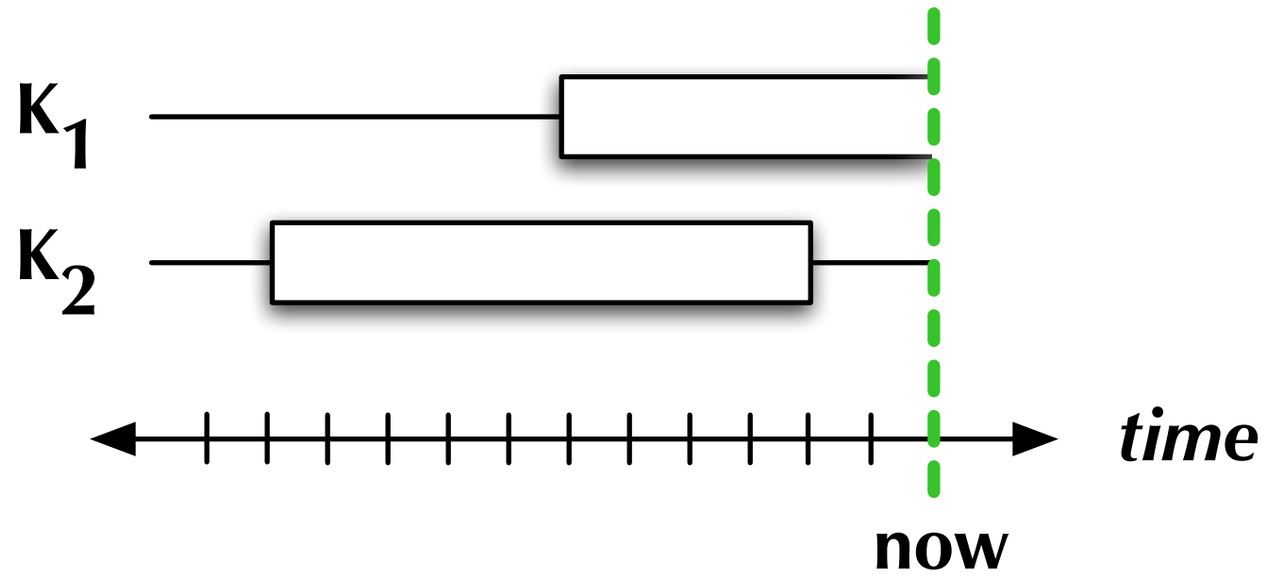
Outline

1. Application-Level Caching
2. TxCache Interface
3. Ensuring Transactional Consistency
4. **Automating Invalidations**
5. Evaluation

Invalidations

What about objects that are still valid?

- don't know their upper validity bound yet!
- represent as open-ended validity intervals



Later, database notifies cache if object changes; cache truncates interval

Invalidation Tags

How to identify which objects changed?

- DB doesn't know which app-level objects are cached

Objects in cache have ***invalidation tags***

- Modified DB to assign invalidation tags to each query
- DB generates list of tags affected by each update
- Cache finds affected objects and updates interval

Invalidation Tags

Inval. tags come from query's access methods

- `TABLE:KEY=VALUE` for queries that use index lookups
- `TABLE:*` for non-indexed queries (rare)

```
SELECT * FROM users WHERE name = 'floyd';

   [result]

   INVALIDATION TAGS users:name=floyd
```

Invalidation Stream

On each update, DB generates affected tags:

- for each tuple affected, one tag per index key

Broadcasts to all cache nodes

- ordered stream, with transaction timestamps

Cache lookups treat unbounded intervals as bounded at last timestamp received

- avoids invalidate & lookup race conditions

Outline

1. Application-Level Caching
2. TxCache Interface
3. Ensuring Transactional Consistency
4. Automating Invalidations
5. **Evaluation**

Evaluation

- How much benefit from adding caching?
- Does using stale data help?
- Does consistency hurt performance?

RUBiS Benchmarks

RUBiS: simulated eBay-like auction site

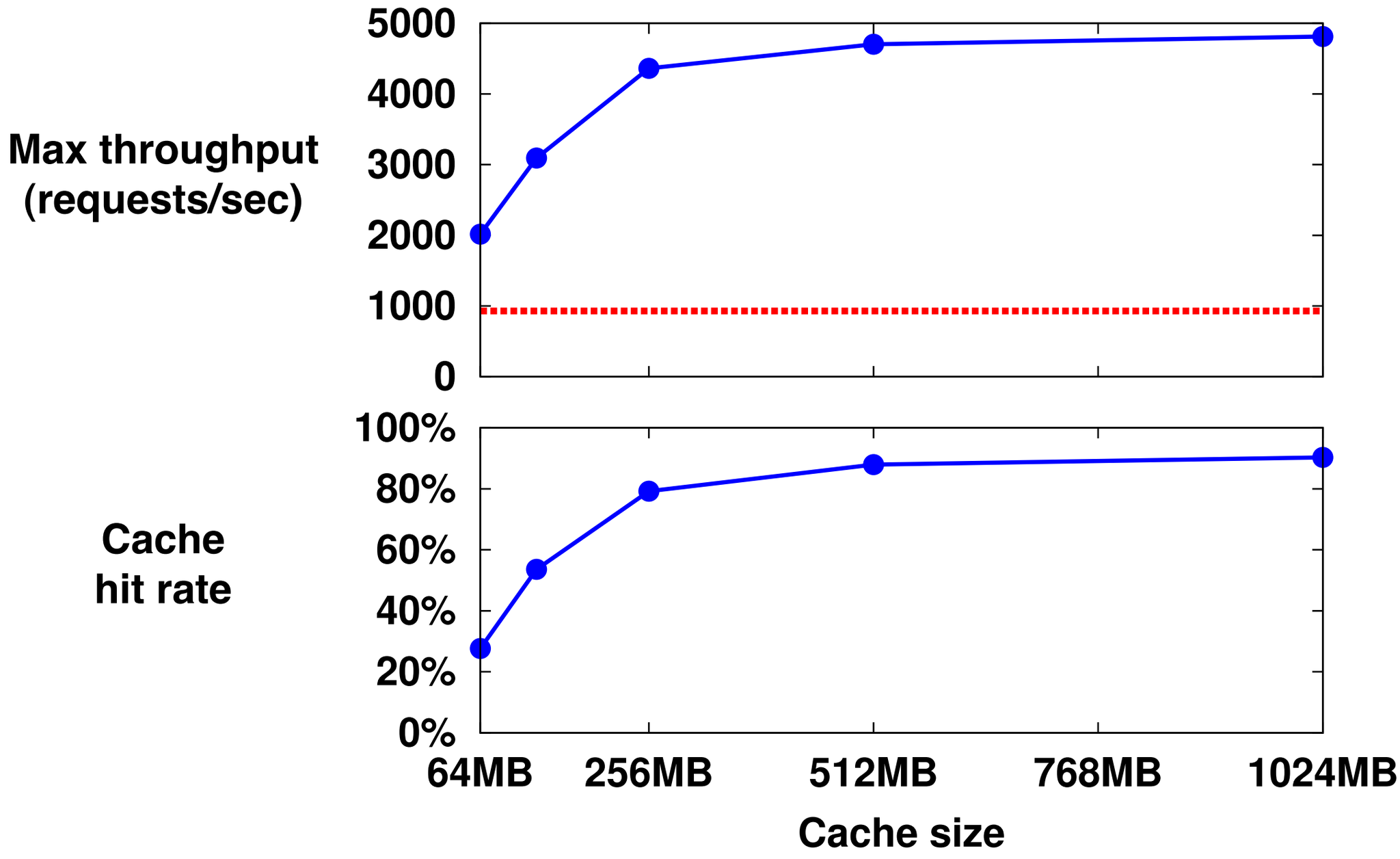
- standard browsing & bidding workload; 85% read-only
- two datasets: 850 MB (in-memory), 6 GB (disk-bound)

All servers 2x 3.20 GHz Xeon, 2 GB RAM

- 1 DB server (modified Postgres 8.2.11)
- 9 frontend/cache servers (Apache 2 / PHP 5)

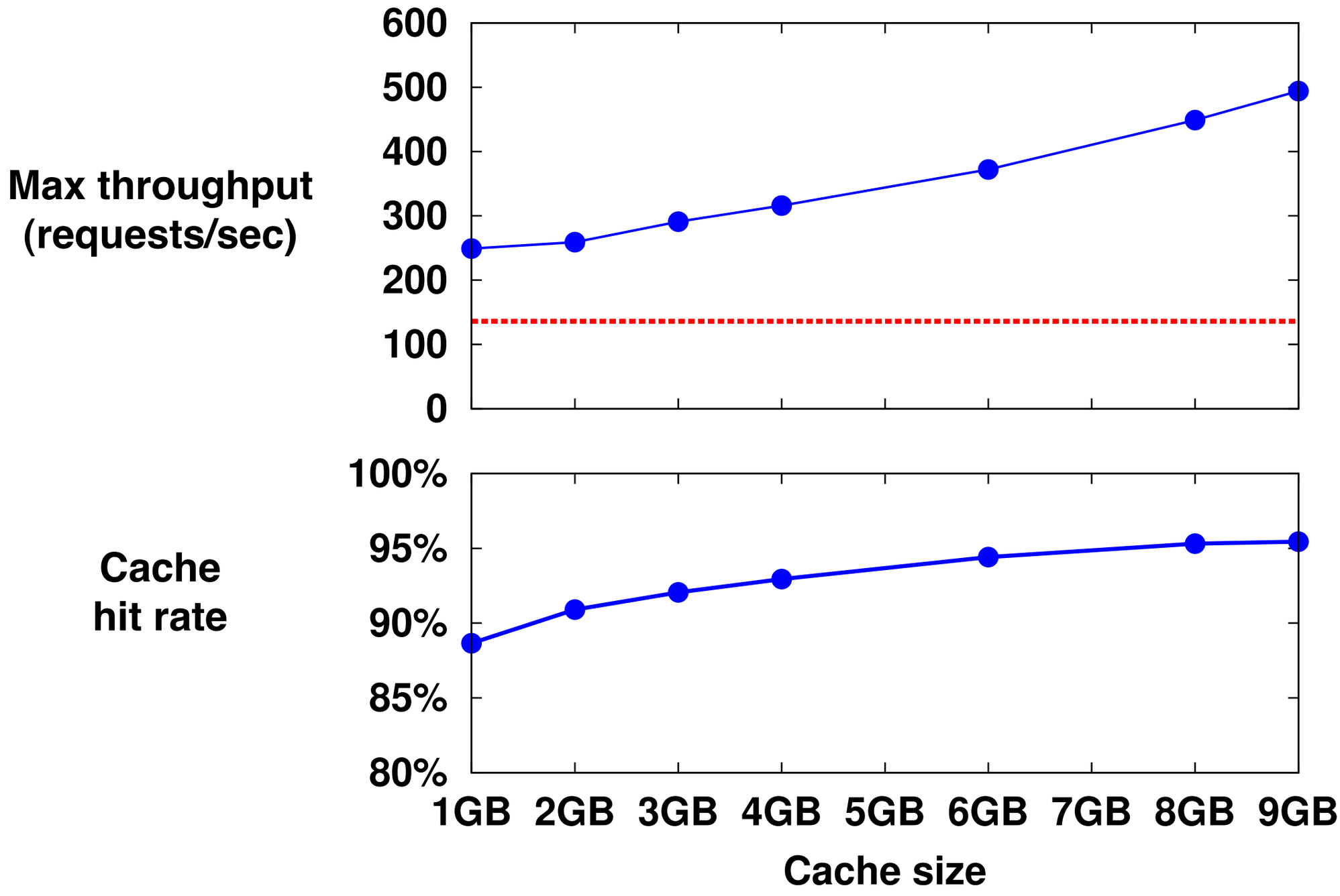
Cache Performance

(in-memory DB; 2 cache nodes)

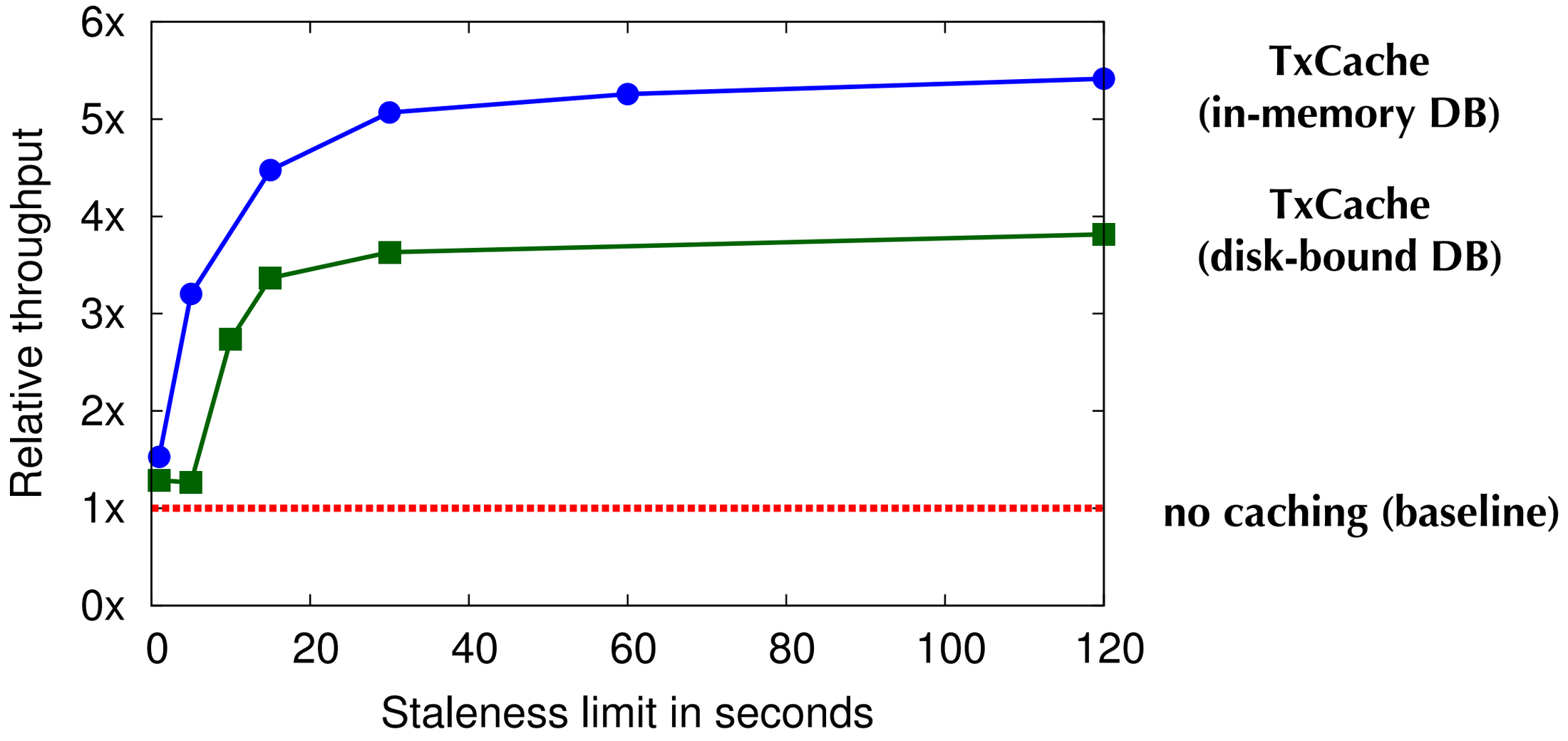


Cache Performance

(disk-bound DB; 8 shared web/cache nodes)



Even A Little Staleness Helps



Costs of Consistency

Cache misses classified as:

- ***compulsory***: data never seen
- ***staleness***: data invalidated & too old to use
- ***capacity***: data was evicted
- ***consistency***: data available but inconsistent w/ prior reads

configuration	consistency misses (% of total misses)
in-memory, 512 MB, 30 s stale	7.8%
in-memory, 512 MB, 15 s stale	5.4%
in-memory, 64 MB, 30 s stale	0.2%
disk-bound, 9 GB, 30 s stale	0.7%

Costs of Consistency

Cache misses classified as:

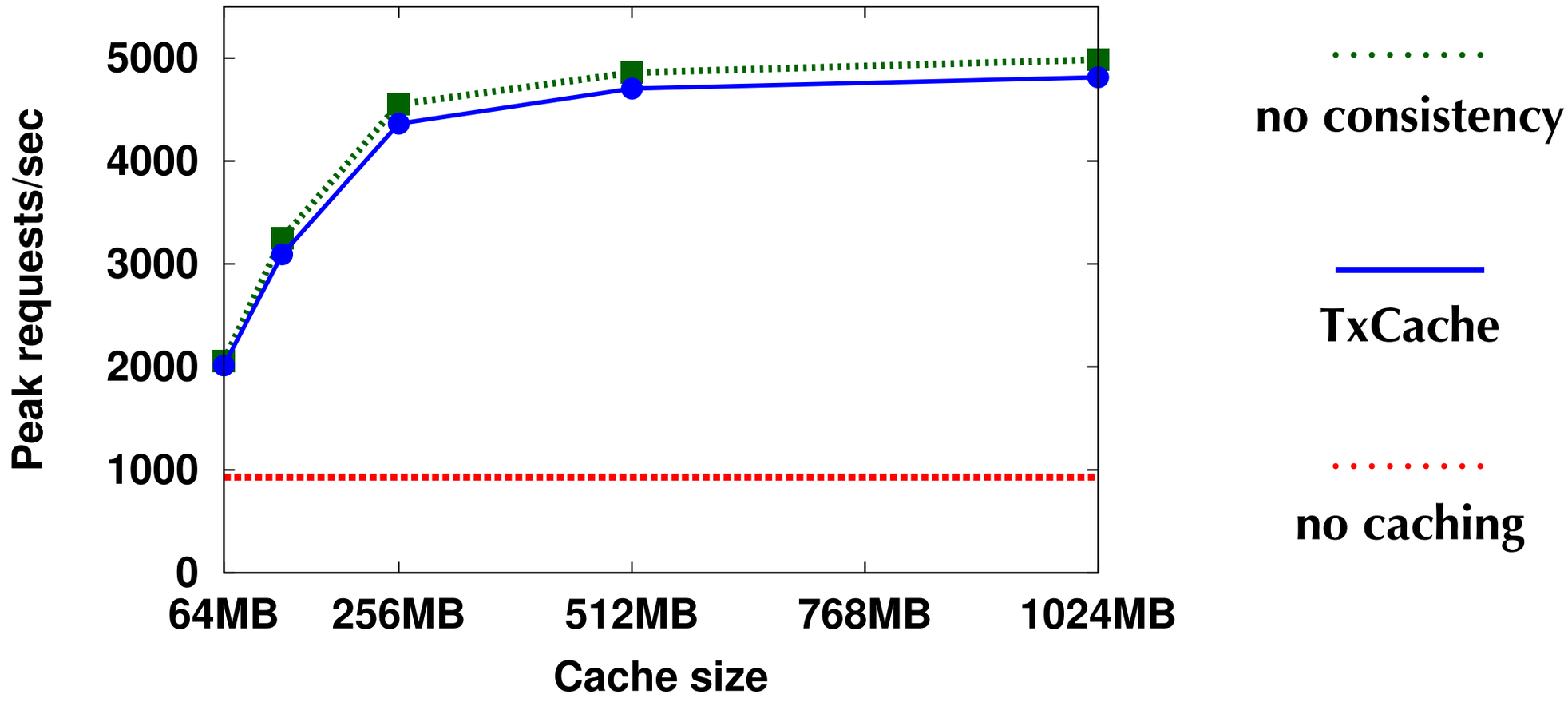
- ***compulsory***: data never seen
- ***staleness***: data invalidated & too old to use
- ***capacity***: data was evicted
- ***consistency***: data available but inconsistent w/ prior reads

} common to other caches

configuration	consistency misses (% of total misses)
in-memory, 512 MB, 30 s stale	7.8%
in-memory, 512 MB, 15 s stale	5.4%
in-memory, 64 MB, 30 s stale	0.2%
disk-bound, 9 GB, 30 s stale	0.7%

Costs of Consistency

Verified experimentally by disabling consistency:
transaction can read any data valid in last 30 sec



Related Work

Application-level caches:

- more flexible than whole-page caches: partial results
- require explicit management by application
- no transactional support (e.g. memcached)
  or transactions only within cache (e.g. JBoss, AppFabric)

Database replication:

- FAS, Ganymed: keep stale replicas with batched updates
- can't apply methods to app-level caching

Conclusion

TxCache: application-layer caching with a simpler programming model

- provides transactional consistency across both cache and database
- automatic management: applications not responsible for lookups, updates, invalidations

New mechanisms:

- consistency ensured by tracking object validity intervals
- automatic database-generated invalidations

Consistency imposes little performance cost