Structured data management on top of large-scale distributed platforms

Ioana Manolescu

Inria

ioana.manolescu@inria.fr

http://pages.saclay.inria.fr/ioana.manolescu

Context: NoSQL systems

- NoSQL = Not Only SQL
- Goal 1: more flexible **data models**
  - Multi-valued attributes; heterogeneous tuples; trees, graphs, lack of types etc.
- Goal 2: **lighter architectures and systems**
  - Fewer constraints, faster development
  - Among the heaviest aspects of data in relational databases: concurrency control
- Goal 3: large-scale **distribution**
- **Can't have ACID at scale: CAP theorem**

Classical DBMS architectures do not cope with large-scale distribution: the CAP theorem

- Eric Brewer, « Symposium on Principles of Distributed Computing », 2000 (conjecture)
- Proved in 2002

- No distributed system can simultaneously provide

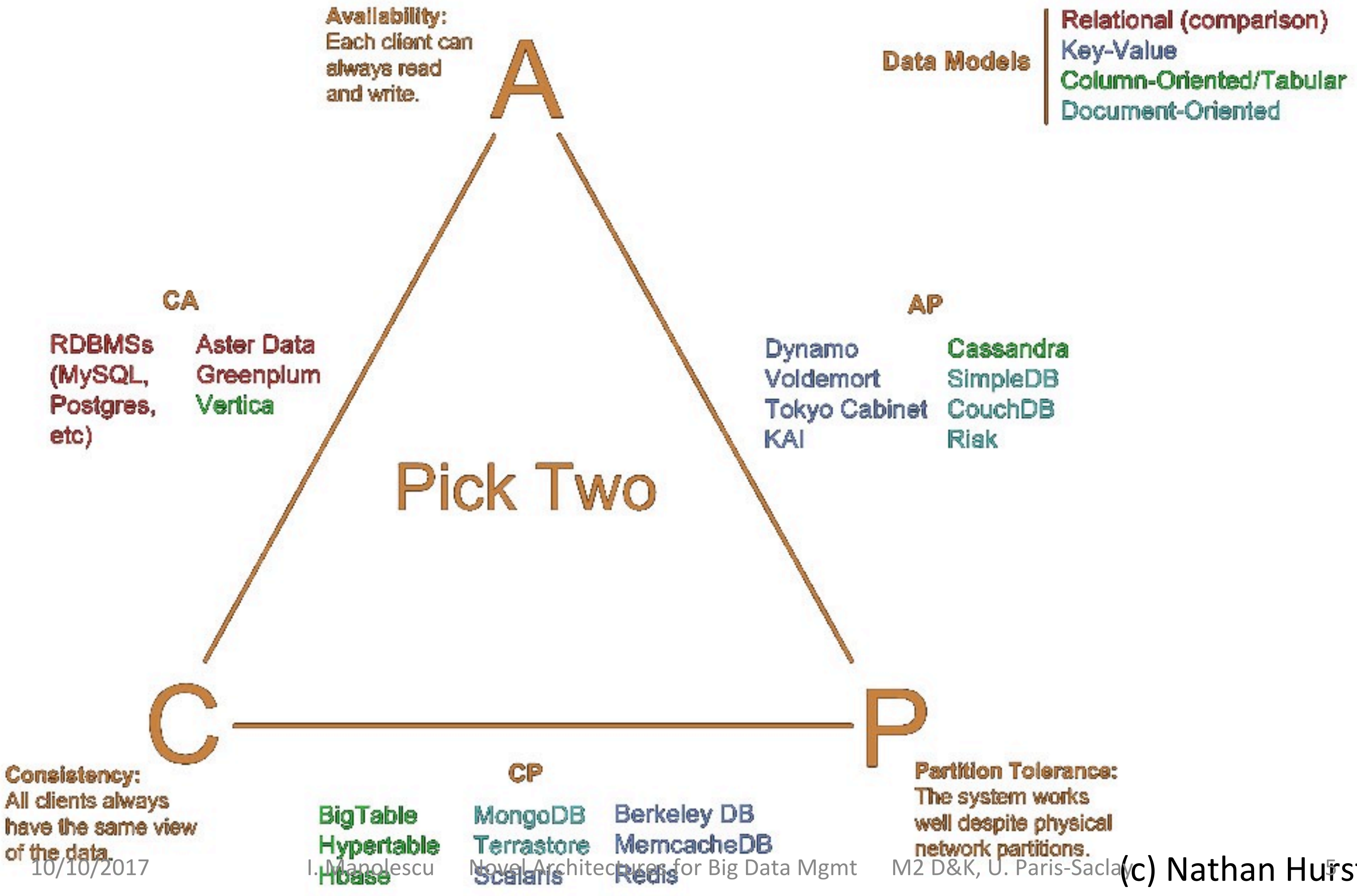
1. **Consistency** (all nodes see the same data at the same time)
2. **Availability** (node failures do not prevent survivors from continuing to operate)
3. **Partition tolerance** (the system continues to operate despite arbitrary message loss)

Some NoSQL systems

Also:

Spark, Hive, Pig, Giraphe…

NoSQL systems vs. CAP theorem



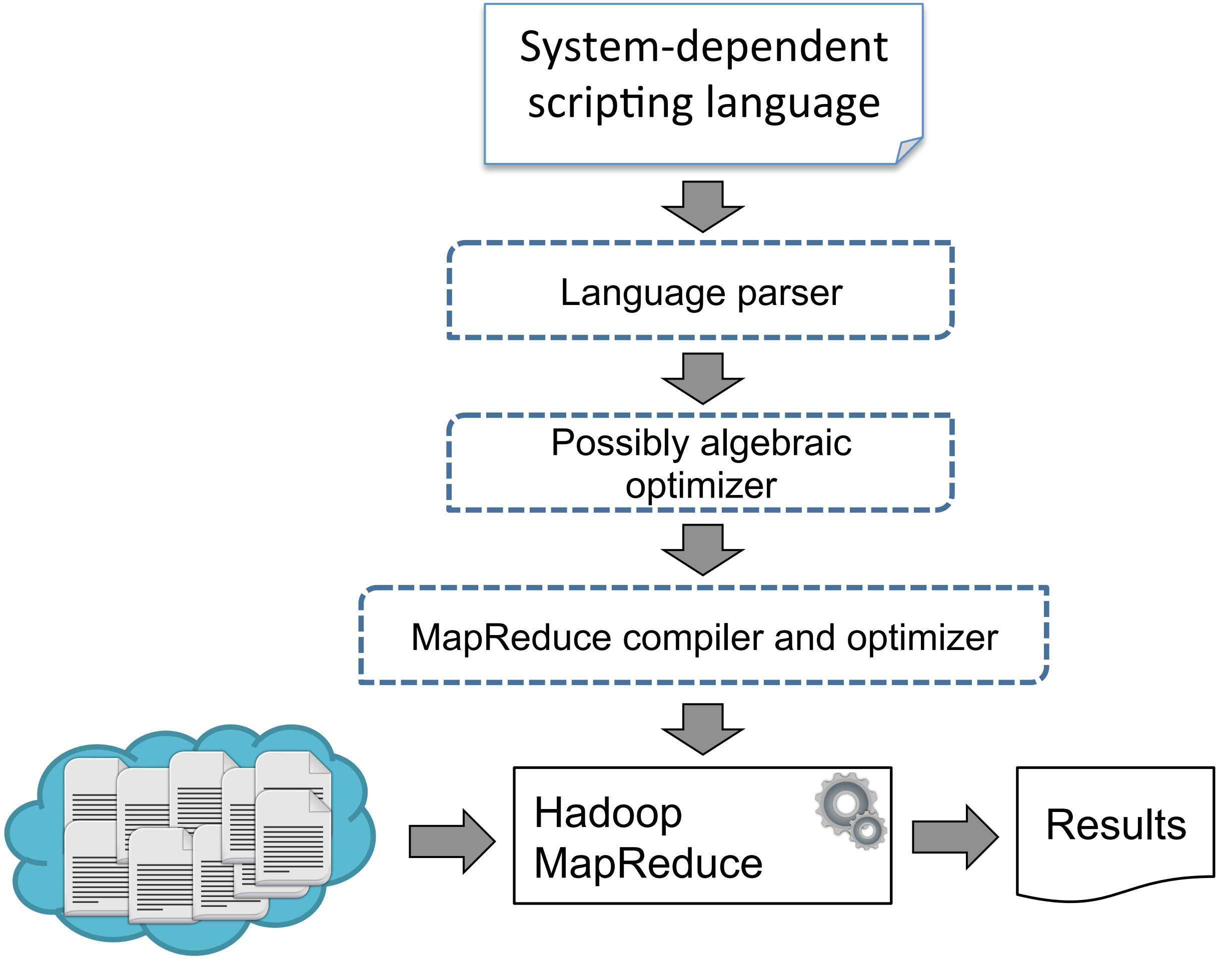
(c) Nathan Hurst

STRUCTURED DATA MANAGEMENT ON TOP OF MAP-REDUCE

Structured DM on top of MapReduce

- We have seen:
  - Techniques for improving **data access selectivity** in a distributed file system (headers; multiple indexes)
  - Algorithms for **implementing operators**: select, project, join
  - **Query optimization** for massively parallel, n-ary joins
- Today:
  - A few highly visible **systems**
  - Some of their mechanisms for **consistency** in a distributed setting

Structured DM on top of MapReduce



Google Bigtable [CDG+06]

- One of the earliest NoSQL systems
- **Goal**: store data of varied form to be used by Google applications:
  - Web indexing, Google Analytics, Finance etc.
- **Approach**:
  - very large, heterogeneous-structure table
- Data model:

Row key → column key → timestamp → value

Different rows can have different columns, each with their own timestamps etc.

Google Bigtable

r1

c0	c1	c4	c7
…	…	…	…

r2

c1	c2	c3	c4	c5	c6
ts11:v1	ts21:v22 ts22:v22	ts31:v31 ts32:v32 ts33:v33	ts41:v41 ts42:v42	ts22:v51	ts61:v61 ts22:v62

Google Bigtable

- **Row key → column key → timestamp → value**
- Rows stored **sorted** in lexicographic order by the key
- Row range dynamically partitioned into **tablets**
  – Tablet = distribution / partitioning unit
- Writes to a row key are atomic
  – row = concurrency control unit
- Access control unit = **column families**
  – Family = typically same-type, co-occurring columns
  – « At most hundreds for each table »
  – E.g. **anchor** column family in Webtable

Apache projects around Hadoop

Hive: relational-like interface on top of Hadoop

- HiveQL language:

CREATE table pokes (foo INT, bar STRING);

SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';

FROM pokes t1 JOIN invites t2 ON (t1.bar = t2.bar)
INSERT OVERWRITE TABLE events SELECT t1.bar, t1.foo, t2.foo;

+ possibility to plug own Map or Reduce function when needed…

Apache projects around Hadoop

- **HBASE:** very large tables on top of HDFS («*goal: billions of rows x millions of columns* »), based on « *sharding* »
- Apache version of Google's BigTable [CDG+06] (used for Google Earth, Web indexing etc.)

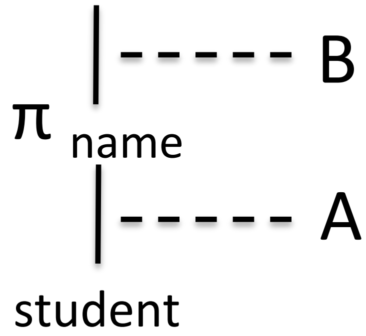
- Main strong points:
  – Fast access to individual rows
  – read/write consistency
  – Selection push-down (~ Hadoop++)
- Does not have: column types, query language, …

Apache projects around Hadoop

PIG: rich dataflow (« SQL + PL/SQL » style) language on top of Hadoop

Suited for many-step data transformations (« extract-transform-load »)

```
A = LOAD 'student' USING PigStorage()
      AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name;
DUMP B;
```



- Flexible data model (~ nested relations)
- Some nesting in the language (< 2 FOREACH ☺)

Apache projects around Hadoop

PIG: rich dataflow (« SQL + PL/SQL » style) language on top of Hadoop

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
DUMP A;
(1,2,3) (4,2,1) (8,3,4) (4,3,3) (7,2,5) (8,4,3)
B = GROUP A BY f1;
DUMP B;
(1,{(1,2,3)}) (4,{(4,2,1),(4,3,3)}) (7,{(7,2,5)})
(8,{(8,3,4),(8,4,3)})
X = FOREACH B GENERATE COUNT(A);
DUMP X;
(1L) (2L) (1L) (2L)
```

|----- X
count(*)
|----- B
Γ_{f1}
|----- A
data

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

r

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)} : B);
STORE F INTO 'S2out';
STORE F INTO 'S3out';
```

45% of the original s_1 + s_2 execution time

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

r

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)} : B);
STORE F INTO 'S2out';
STORE F INTO 'S3out';
```

Join

45% of the original s_1 + s_2 execution time

PigLatin: repeated execution of some computations

s_1

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name, B BY user;
D = FOREACH C GENERATE name, address, time;
STORE D INTO 'S1out';
E = JOIN A BY name LEFT, B BY user;
STORE E INTO 'S2out';
```

s_2

```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = JOIN A BY name LEFT, B BY user;
STORE C INTO 'S3out';
```

r

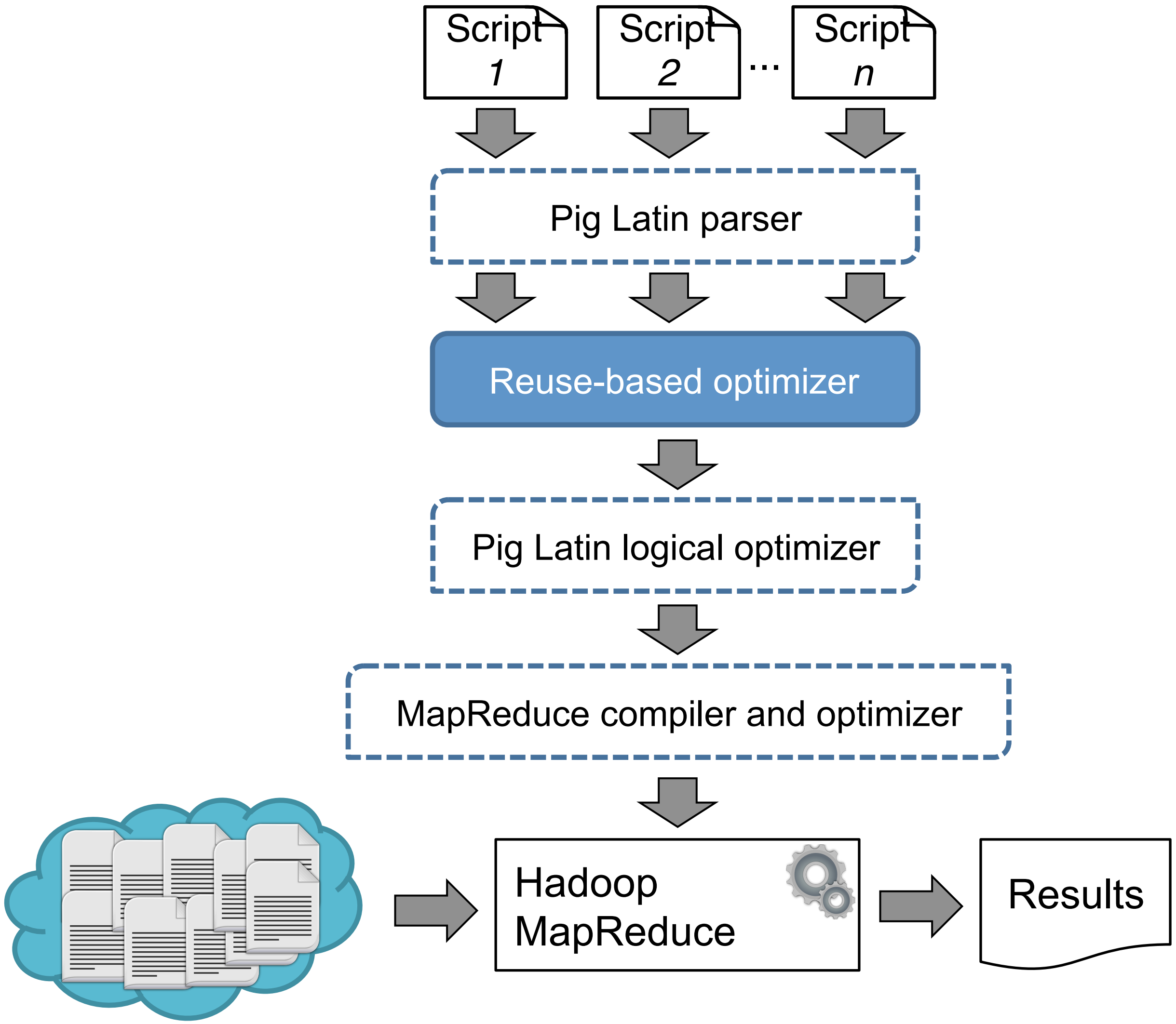
```
A = LOAD 'users' AS (name, address);
B = LOAD 'page_views' AS (user, www, time);
C = COGROUP A BY name, B BY user;
D = FOREACH C GENERATE flatten(A), flatten(B);
E = FOREACH D GENERATE name, address, time;
STORE E INTO 'S1out';
F = FOREACH C GENERATE flatten(A), flatten (isEmpty(B) ? {(null,null,null)} : B);
STORE F INTO 'S2out';
STORE F INTO 'S3out';
```

Join

Left outer join

45% of the original s_1 + s_2 execution time

Reuse-based optimizer within Pig [CCH+16]



Optimizer:

- **Translates** PigLatin programs into nested relational algebra for bags
- Applies equivalence laws to **identify repeated subexpressions**
- **Replaces** all but one of the subexpressions, **reuses** the result of the last
- Reduced execution time by x4

Apache projects around Hadoop

Cassandra

- (Large, distributed) relations on top of Hadoop
- Some nesting (a field can be a collection); indexes; SQL-like access rights
- Queries: select, project. No join ☺

Table **songs**:

id	song_order	album	artist	song_id	title
62c36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Rojo
62c36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside Woman Blues
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Stereo
62c36092...	1	Tres Hombres	ZZ Top	a3e64f8f...	La Grange

ALTER TABLE songs *ADD tags set<text>*;

UPDATE songs SET tags = tags + {'2007'} WHERE id = 8a172618...;
UPDATE songs SET tags = tags + {'covers'} WHERE id = 8a172618...;
UPDATE songs SET tags = tags + {'1973'} WHERE id = a3e64f8f-...;

SELECT id, tags from songs;

id	tags
7db1a490-5878-11e2-bcfd-0800200c9a66	{rock}
a3e64f8f-bd44-4f28-b8d9-6938726e34d4	{blues, 1973}
8a172618-b121-4136-bb10-f665cfc469eb	{2007, covers}

Spanner: A More Recent Google Distributed Database [CD+12]

- A few **Universes** (e.g. one for production, one for testing)
- Universe = set of zones
  - **Zone** = unit of administrative deployment
  - One or several zones in a datacenter
  - 1 zone = 1 **zone master** + 100s to 1000s of **span servers**
  - The zone master assigns data to span servers
  - Each span servers answers client requests
  - Each span server handles 100 to 1000 tablets
- **Tablet** = { key → timestamp → string }
- **Table** = set of tablets.

More on the Spanner data model

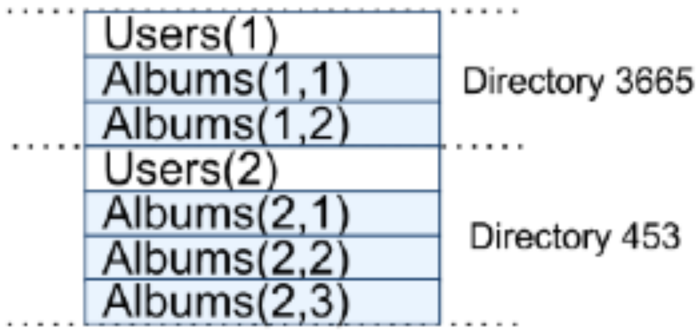
- Basic: **key → timestamp → value**
- **Directory** (or **bucket**): set of contiguous keys that share a common prefix
  - Data moves around by the bucket/directory
- On top of the basic model, applications see a **surface** **relational** **model**
  - Rows x columns (tables with a **schema**)
  - **Primary keys**: each table must have one or several primary-key columns

Spanner tables

- Tables can be organized in **hierarchies**
  - Tables whose primary key **extends the key of the parent** can be stored **interleaved** with the parent
  - Example: photo album metadata organized first by the user, then by the album

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



Spanner replication

- Used **for very high-availability** storage
- Store data with a **replication** factor (3 to 5)
- Applications can control:
  - Which datacenters control which data
  - How far data is from users (to control read latency)
  - How far replicas are from each other (to control write latency)
  - How many replicas are maintained
- Concurrency control relies on a **global timestamp mechanism** called « TrueTime » (see next)

Spanner TrueTime service

- TT.now() returns **a Ttinterval [earliest; latest]**
  - Uncertainty interval made explicit
  - The interval is guaranteed to contain the absolute time during which TT.now() was invoked
  - TrueTime clients **wait** to avoid the uncertainty
- Based on GPS and atomic clocks
  - Implemented by a set of **time master machines** per datacenter and a **timeslave daemon** per machine
  - Every daemon polls a variety of masters to **reduce vulnerability** to
    - Errors from a single master
    - Attacks

Spanner consistency guarantees

- Linearizability:
  If transaction T1 commits before T2 starts
  Then the commit timestamp of T1 is guaranteed to be smaller than the commit timestamp of T2

→ globally meaningful commit timestamps

→ globally-consistent reads across the database at a timestamp

May not read the *last* version, but one from 5-10 seconds ago! (Last globally committed version.)

Spanner consistency guarantees

- Linearizability:
  If transaction T1 commits before T2 starts
  Then the commit timestamp of T1 is
  gu … it
  tin

→ gl … s

→ gl … abase

at a …

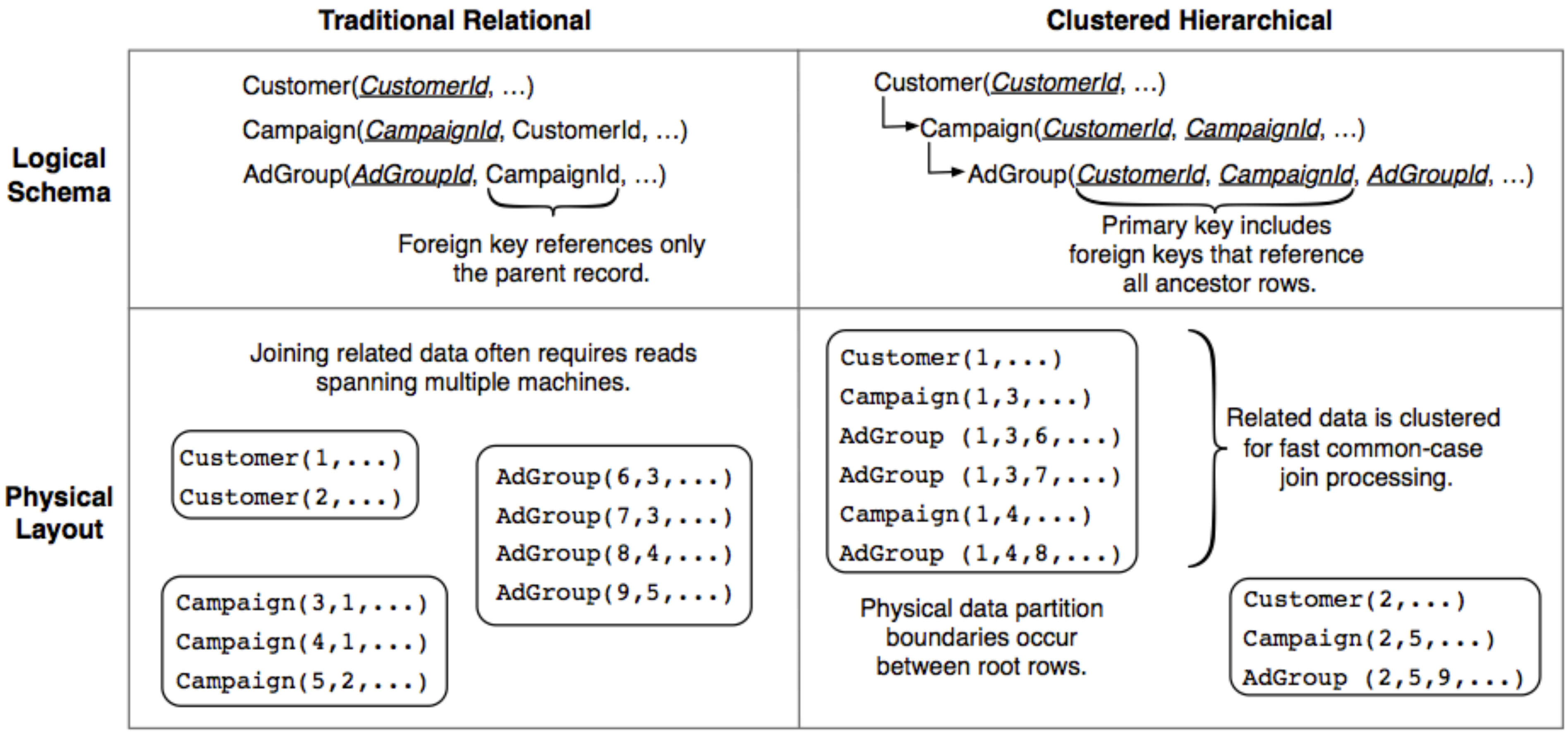
> « Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems it brings. We **believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.** »

F1: Distributed Database from Google [SVS+13]

- Built on top of Spanner
- Goals:
  - Scalability, availability
  - Consistency (almost ACID)
  - Usability (= full SQL + transactional indexes etc.)
- F1 from genetics « Filial 1 Hybrid » (cross mating of very different parental types)
  - F1 is a hybrid between relational DBs and scalable NoSQL systems

F1 data model

- Surface model: relational
- Storage: Clustered, inlined table hierarchies (Spanner)



Transactions in F1

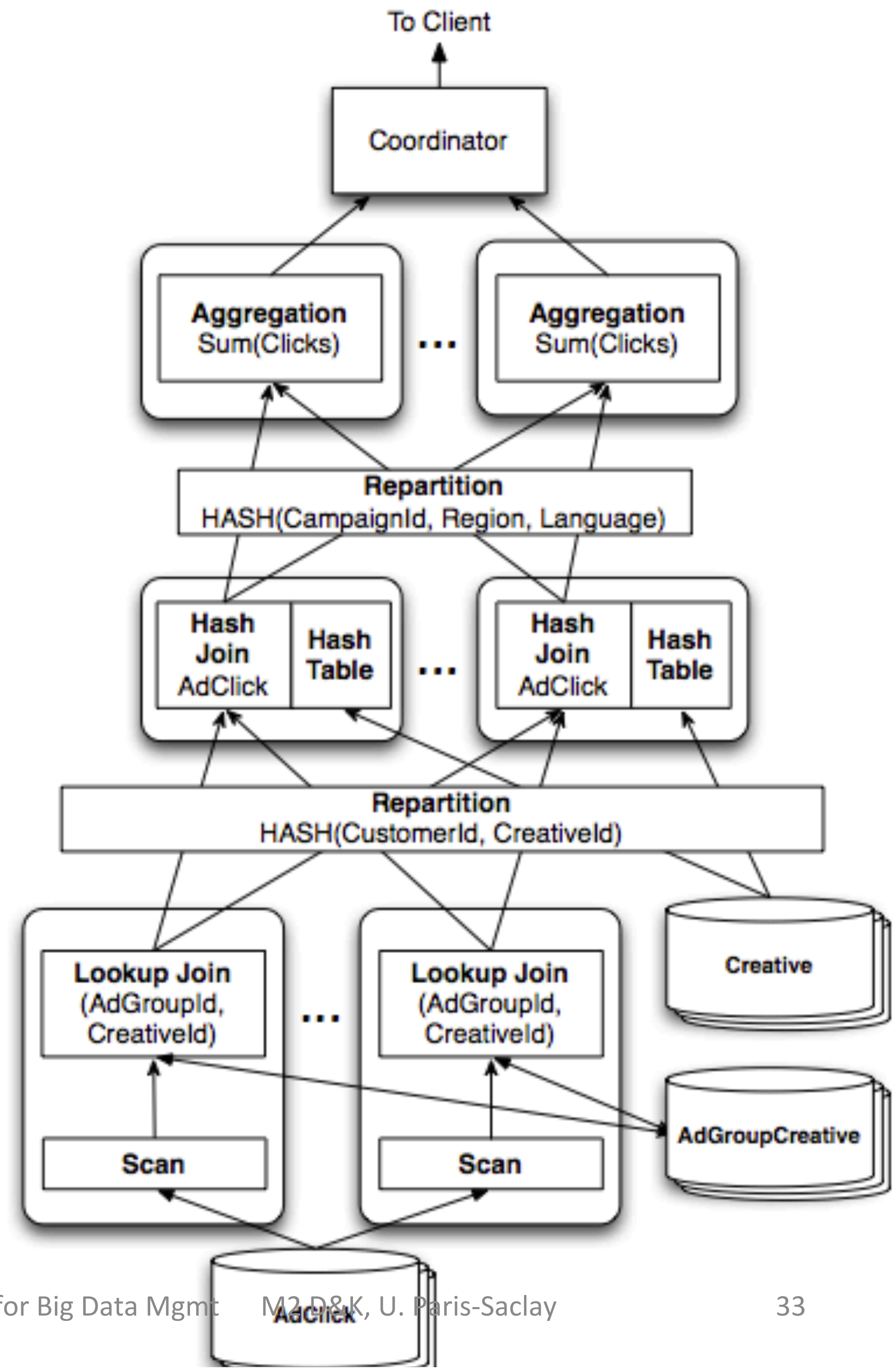
- **Snapshot** (read-only) transactions (no locks)
  - Read at Spanner's global safe timestamp, typically 5-10 seconds old, from a local replica
  - Default for SQL and MapReduce. All clients see the same data at the same timestamp.
- **Pessimistic** transactions (provided by Spanner)
  - Shared or exclusive locks; may abort
- **Optimistic** transactions
  - Read phase (no lock), then short write phase
  - Each row has last modification timestamp
  - To commit optimistic T1, F1 creates a short pessimistic T2 which attempts to read all of T1's rows. If T2 has a different version than T1, then T1 is aborted. Otherwise, T1 commits.

More on transactions in F1

- Benefits of optimistic transactions:
  - Reads never hold locks, never conflict with writes
    - Avoid performance drawback when a read runs for too long or aborts
    - Can run for a long time without hurting performance
- Self-contained: can be retried (after abort) at the F1 server, hiding transient Spanner errors
  - Pessimistic transactions cannot be retried at the server, because they require re-running client operations that took locks
- Drawbacks:
  - Concurrency control through last modif timestamp only works for existing rows → insertion phantoms
    - The same transaction may get different results in two successive reads of the same data
  - Low throughput if high contention as many transactions will abort (pessimistic ones will also abort in this case).

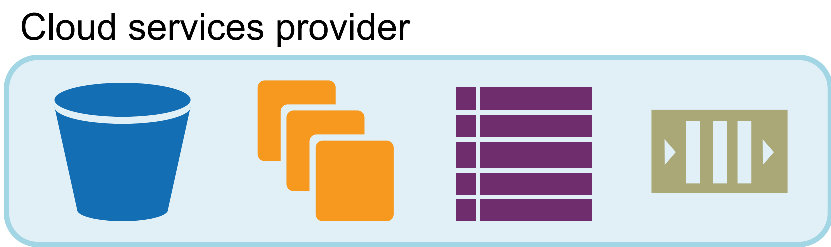
Query optimization in F1

```
SELECT agcr.CampaignId, click.Region,
       cr.Language, SUM(click.Clicks)
FROM AdClick click
  JOIN AdGroupCreative agcr
    USING (AdGroupId, CreativeId)
  JOIN Creative cr
    USING (CustomerId, CreativeId)
WHERE click.Date = '2013-03-23'
GROUP BY agcr.CampaignId, click.Region,
         cr.Language
```



STRUCTURED DATA MANAGEMENT ON TOP OF CLOUD PLATFORMS

Structured data management in cloud platforms



- They offer:
  - Distributed file system
  - Virtual machines
  - Key-value store
  - Distributed message queues

- We need: an architecture for scalable management of complex-structure data (e.g., XML documents, RDF graphs)
  - **Idea**: **performance**- and **cost**-efficient indexing

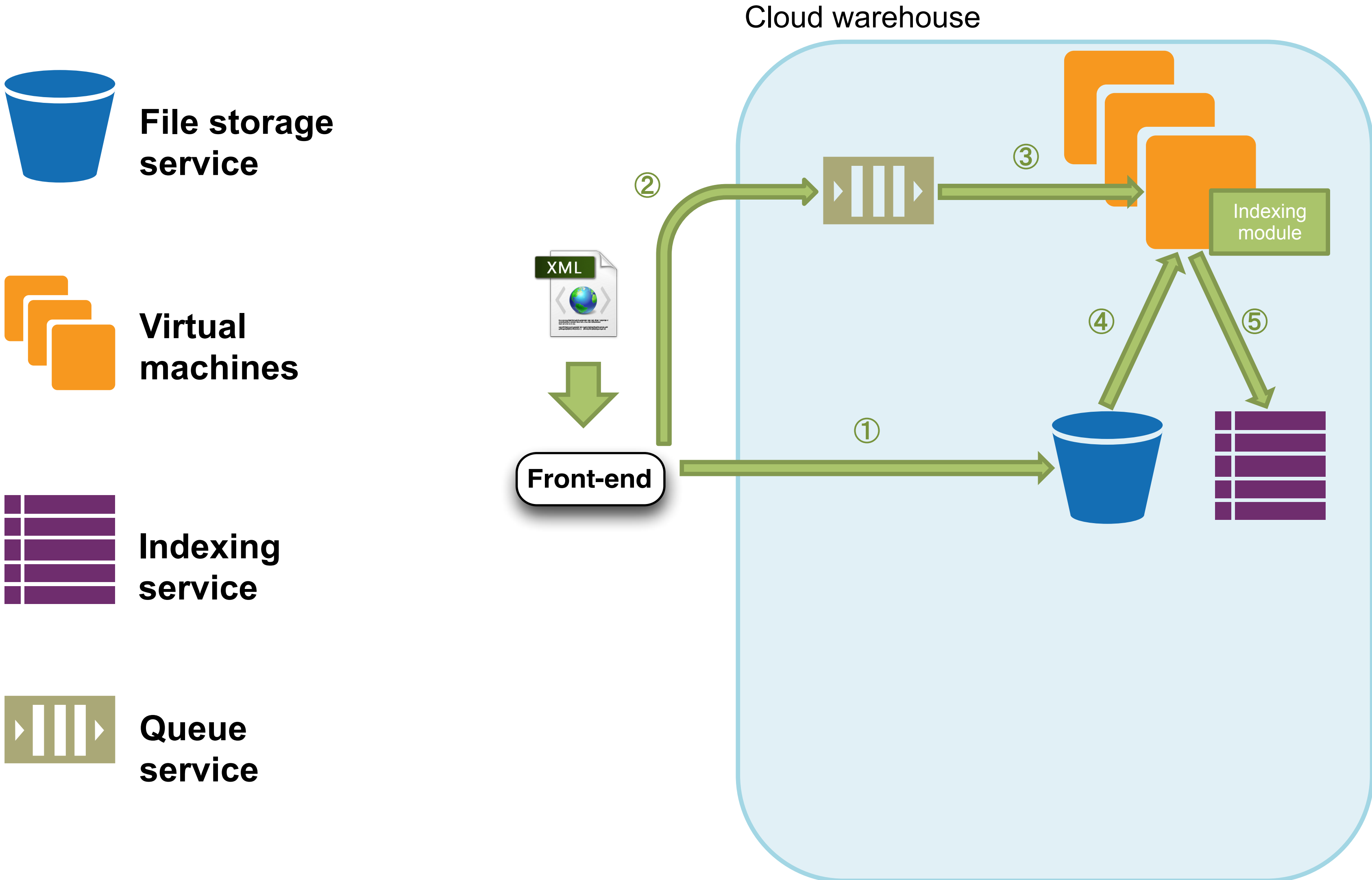
Cloud services

	amazon web services™	Google Cloud Platform	Windows Azure
File storage service	Amazon Scalable Storage Service (S3)	Google Cloud Storage	Windows Azure BLOB Storage
Virtual machines	Amazon Elastic Compute Cloud (EC2)	Google Compute Engine	Windows Azure Virtual Machines
Indexing service	Amazon DynamoDB	Google High Replication Datastore	Windows Azure Tables
Queue service	Amazon Simple Queue Service (SQS)	Google Task Queues	Windows Azure Queues

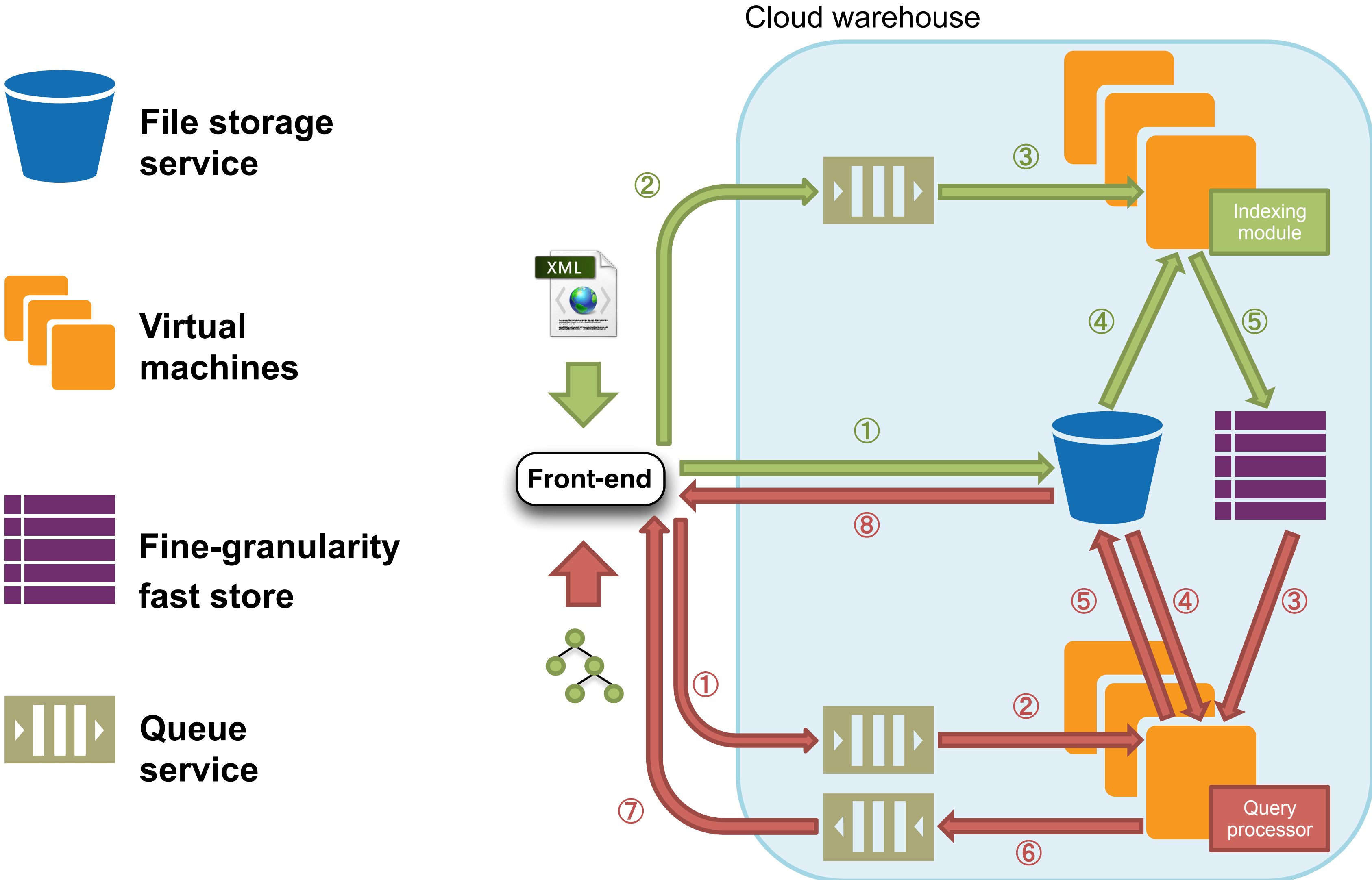
Cloud services

	Amazon web services	Google Cloud Platform	Windows Azure
File storage service	Amazon Scalable Storage Service (S3)	Google Cloud Storage	Windows Azure BLOB Storage
Virtual machines	Amazon Elastic Compute Cloud (EC2)	Google Compute Engine	Windows Azure Virtual Machines
Indexing service	Amazon DynamoDB	Google High Replication Datastore	Windows Azure Tables
Queue service	Amazon Simple Queue Service (SQS)	Google Task Queues	Windows Azure Queues

AMADA: fine-granularity XML indexing in the Amazon cloud [CCM13]



AMADA architecture



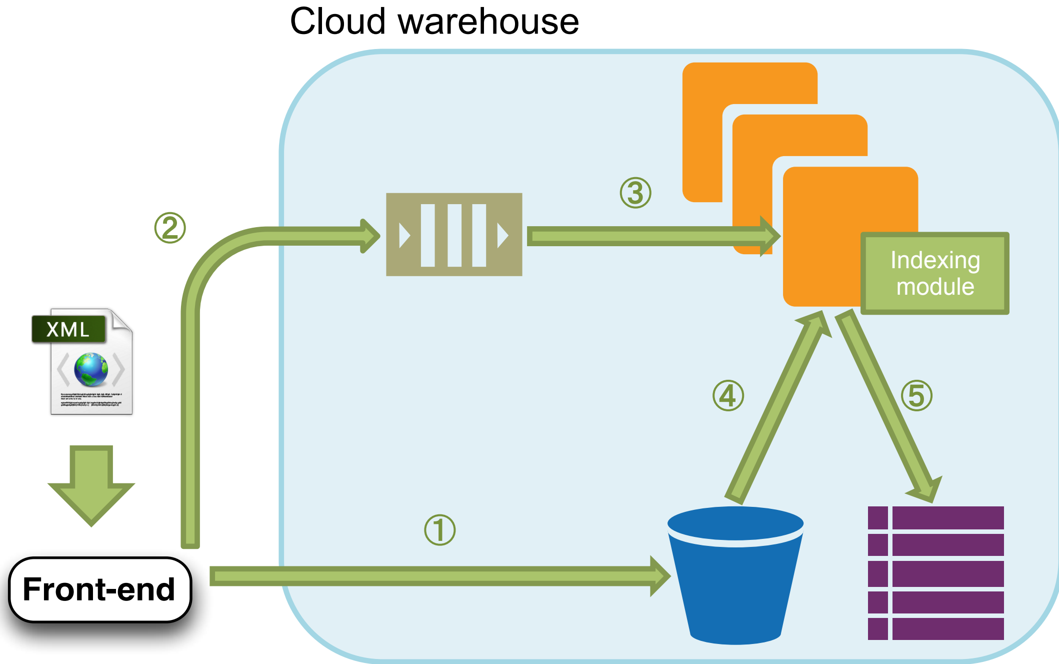
Indexing and storage costs

File storage service

Virtual machines

Fine-granularity fast store

Queue service



Indexing cost depends on:
- Documents set (*D*)
- Indexing strategy (*I*)

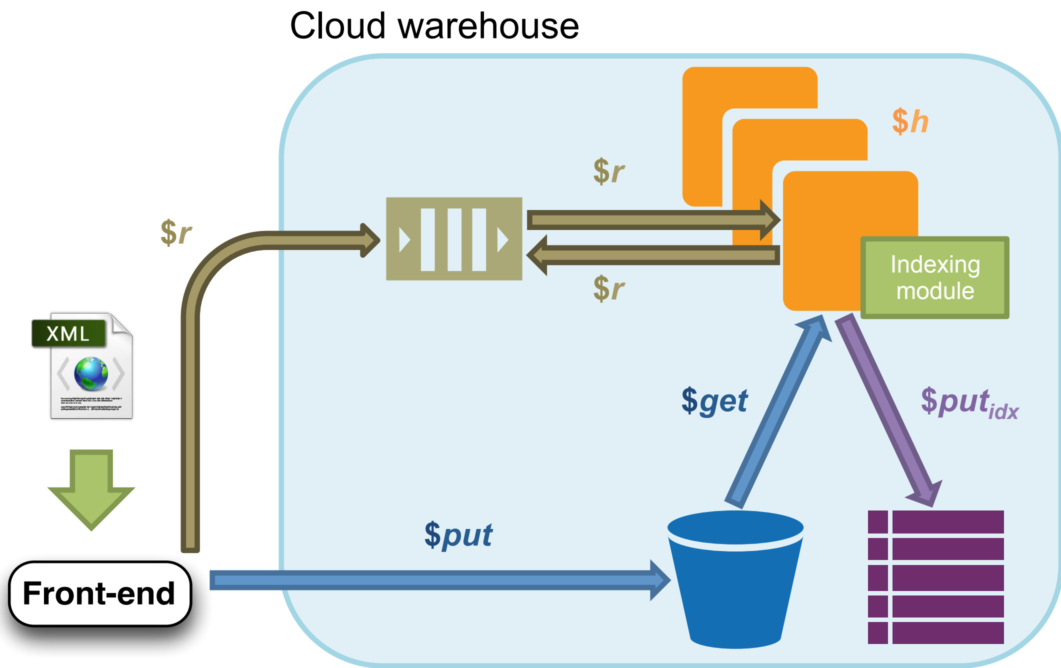
Indexing and storage costs

File storage service

Virtual machines

Fine-granularity fast store

Queue service



Storage cost

Querying cost

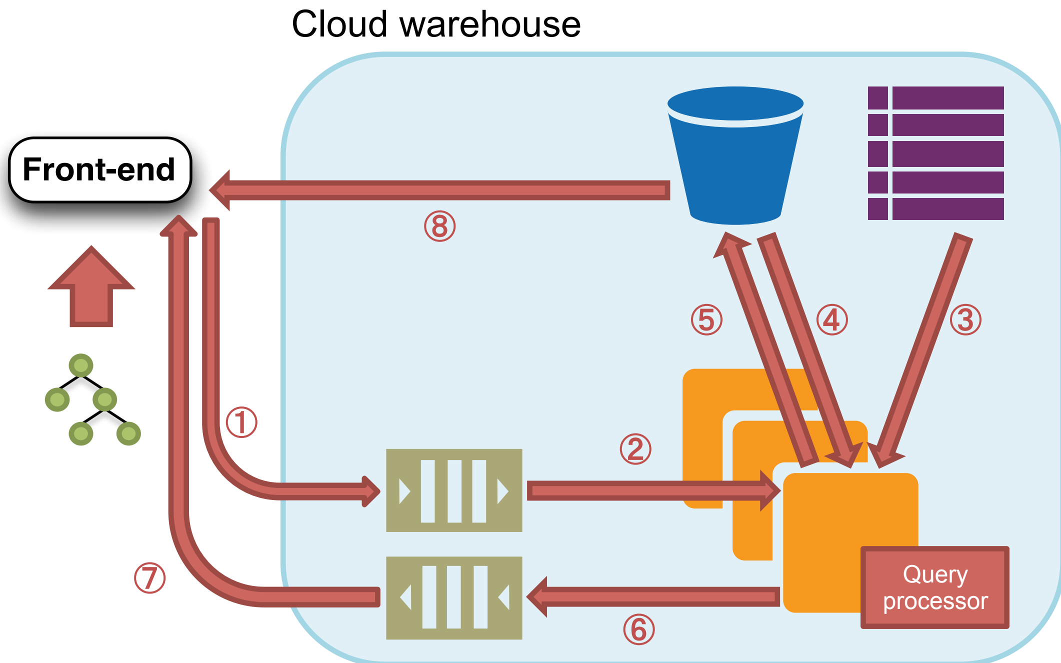
File storage service

Virtual machines

Fine-granularity fast store

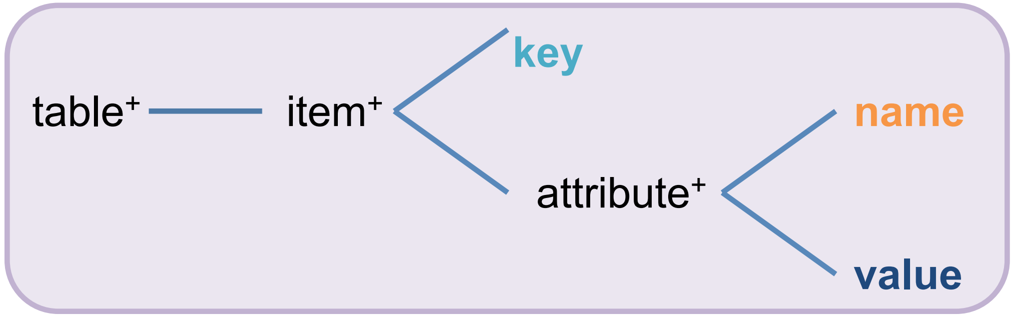
Queue service

Querying cost depends on:
- Query (*q*)
- Documents set (*D*)
- Indexing strategy (*I*)



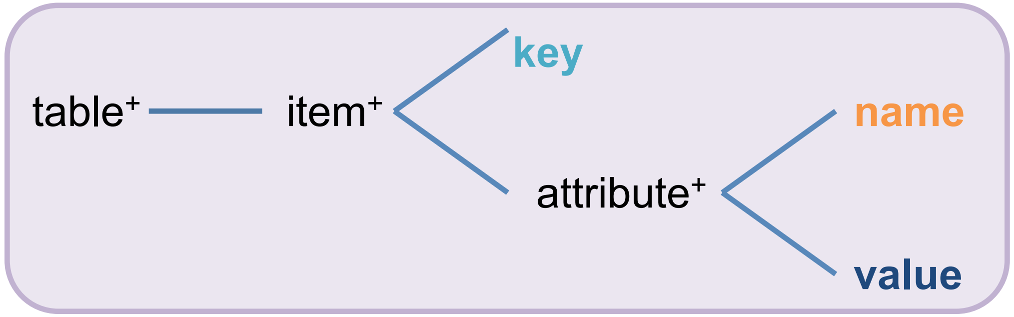
Indexing strategies

DynamoDB data model



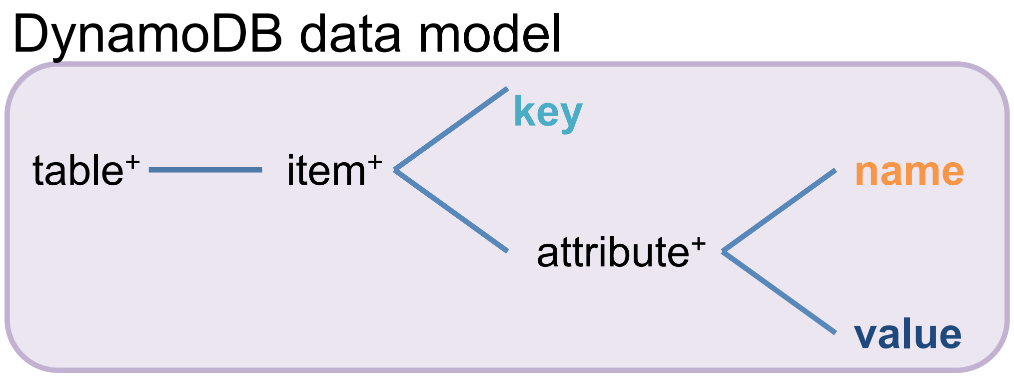
Indexing strategies

DynamoDB data model



Indexing strategy *I*: Function associating (**key**,(**name**,**value**)$^+$)$^+$ to a document

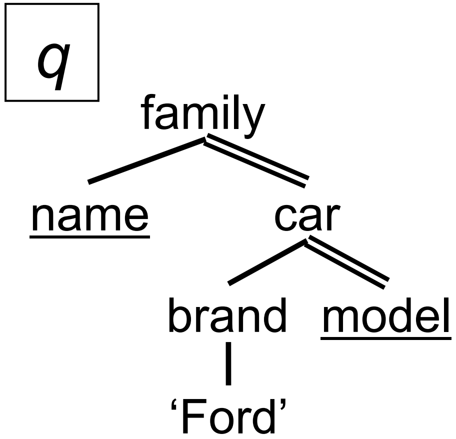
Indexing strategies

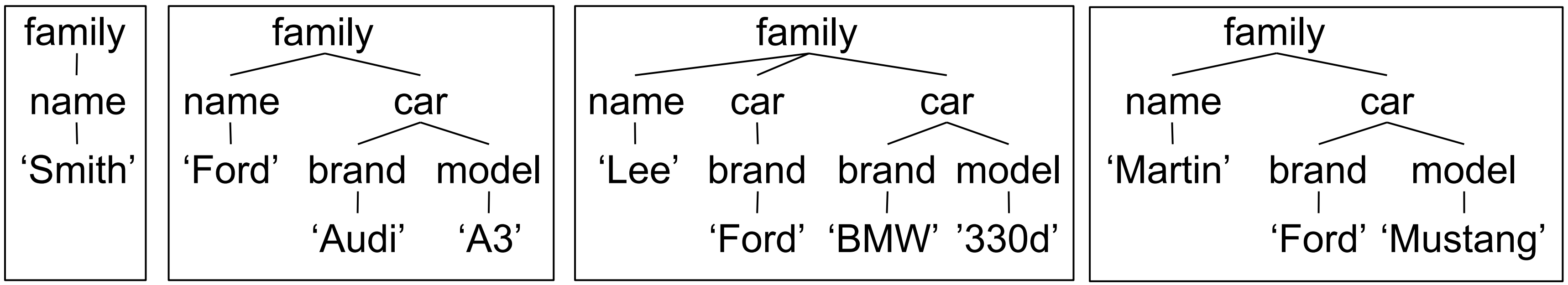


Indexing strategy *I*: Function associating (**key**,(**name**,**value**)$^+$)$^+$ to a document

- Four indexing strategies with different properties:
  - *Label-URI (LU)*
  - *Label-URI-Path (LUP)*
  - *Label-URI-ID (LUI)*
  - *Label-URI-Path/Label-URI-ID (2LUPI)*

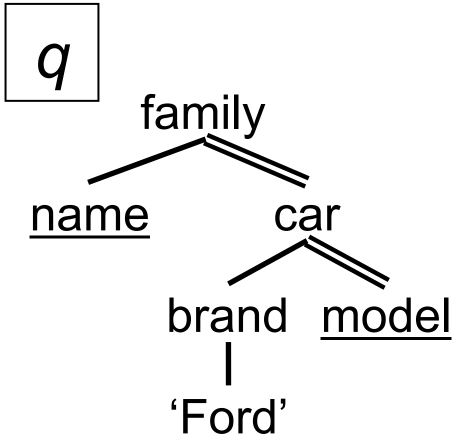
XML indexing: example





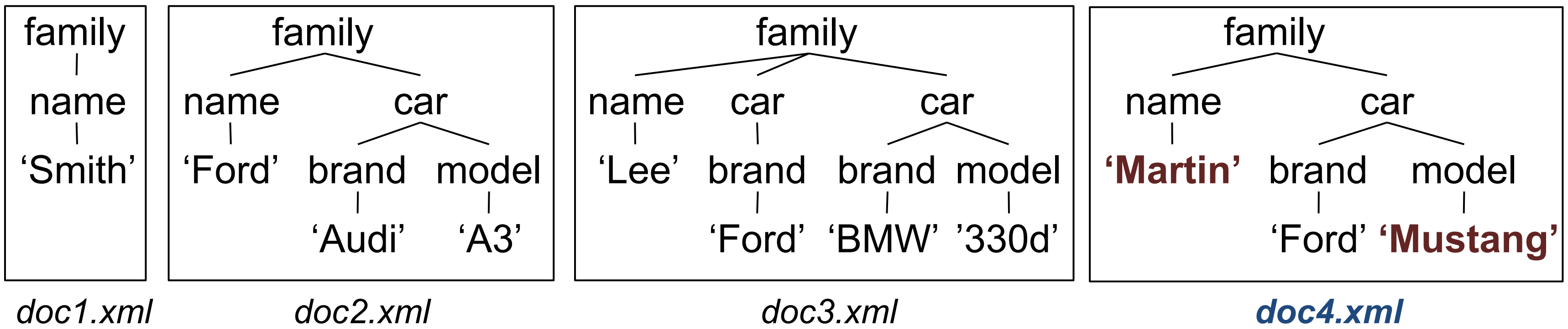
doc1.xml *doc2.xml* *doc3.xml* *doc4.xml*

XML indexing: example



Result:

doc4.xml	**Martin**	**Mustang**



Label-URI (LU) strategy

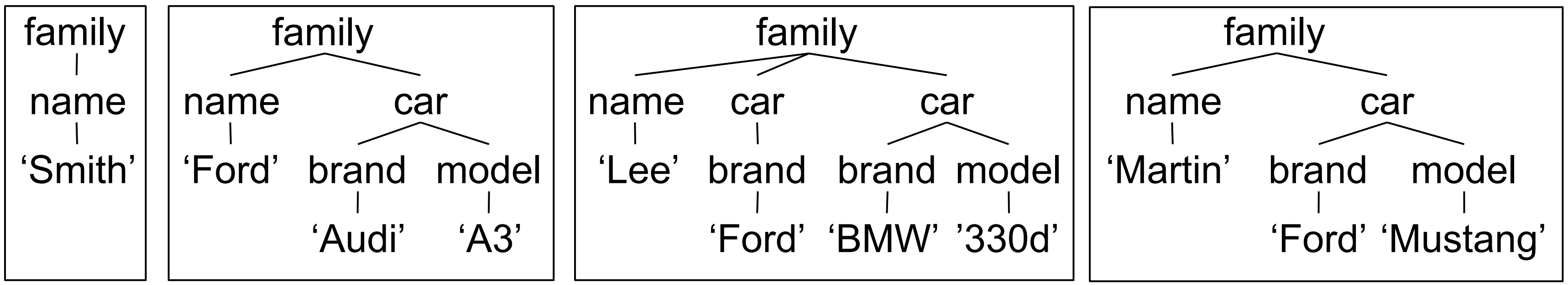
Index:

efamily	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø	Ø

ename	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø	Ø

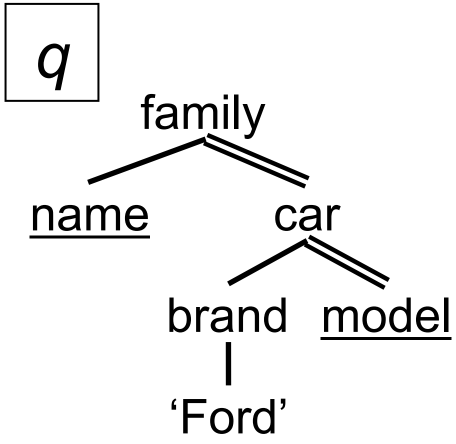
wFord	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø

…



doc1.xml *doc2.xml* *doc3.xml* *doc4.xml*

Label-URI (LU) strategy



Index:

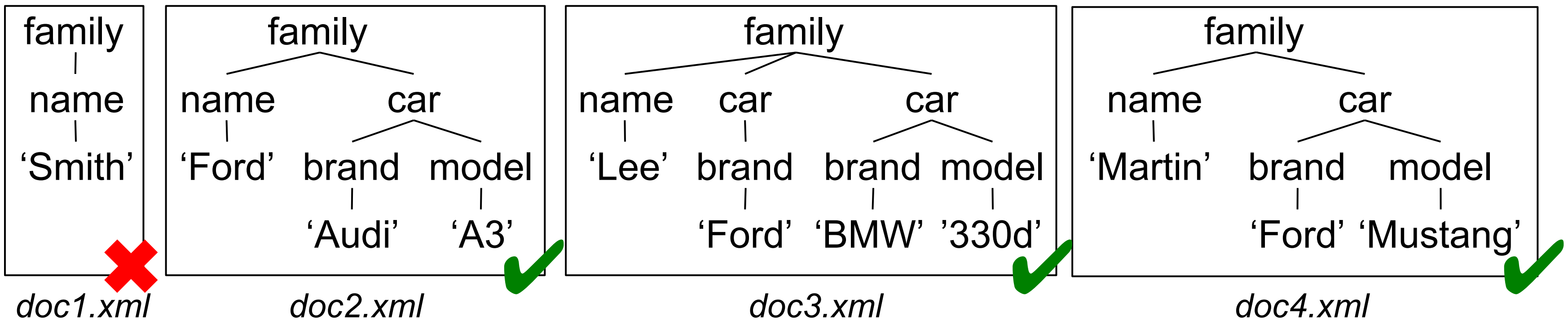
efamily	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø	Ø

ename	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø	Ø

wFord	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø

...

Look-up: Intersection of URI sets associated to each query node



Label-URI-ID (LUI) strategy

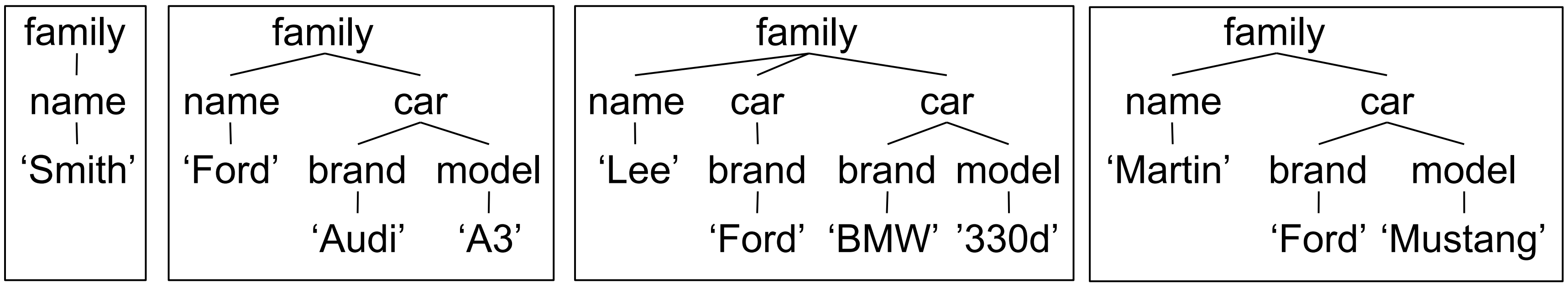
Index:

	doc1.xml	doc2.xml	doc3.xml	doc4.xml
*e*family	[1 3 0]	[1 8 0]	[1 11 0]	[1 8 0]

	doc1.xml	doc2.xml	doc3.xml	doc4.xml
*e*name	[2 2 1]	[2 2 1]	[2 2 1]	[2 2 1]

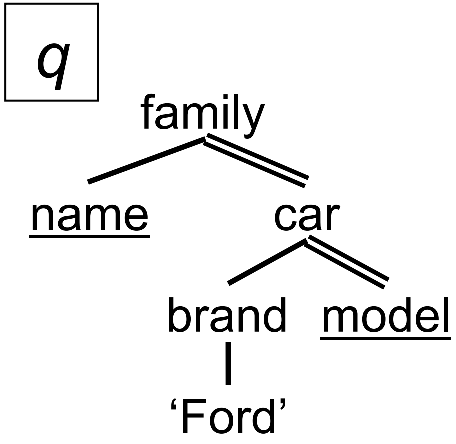
	doc2.xml	doc3.xml	doc4.xml
*w*Ford	[3 1 2]	[6 3 3]	[6 3 2]

…



doc1.xml *doc2.xml* *doc3.xml* *doc4.xml*

Label-URI-ID (LUI) strategy



Index:

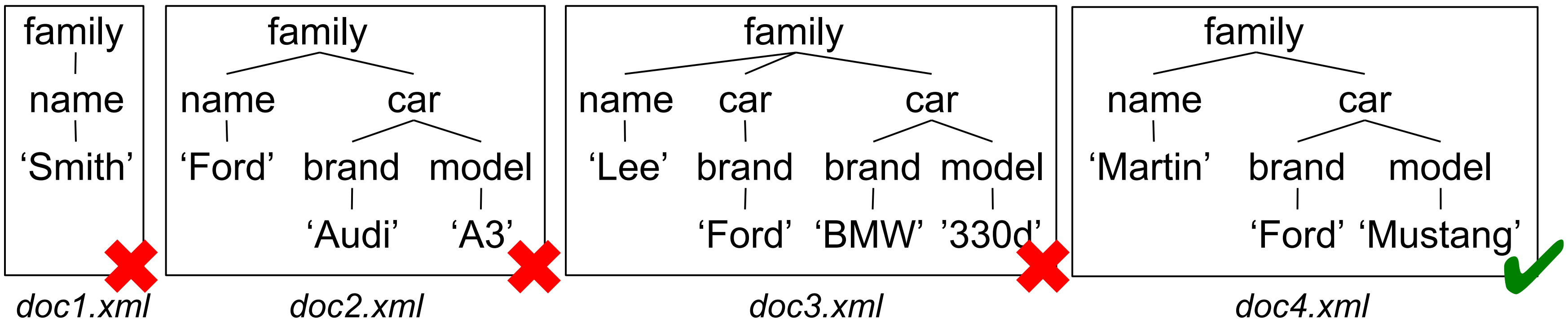
efamily	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	[1 3 0]	[1 8 0]	[1 11 0]	[1 8 0]

ename	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	[2 2 1]	[2 2 1]	[2 2 1]	[2 2 1]

wFord	doc2.xml	doc3.xml	doc4.xml
	[3 1 2]	[6 3 3]	[6 3 2]

…

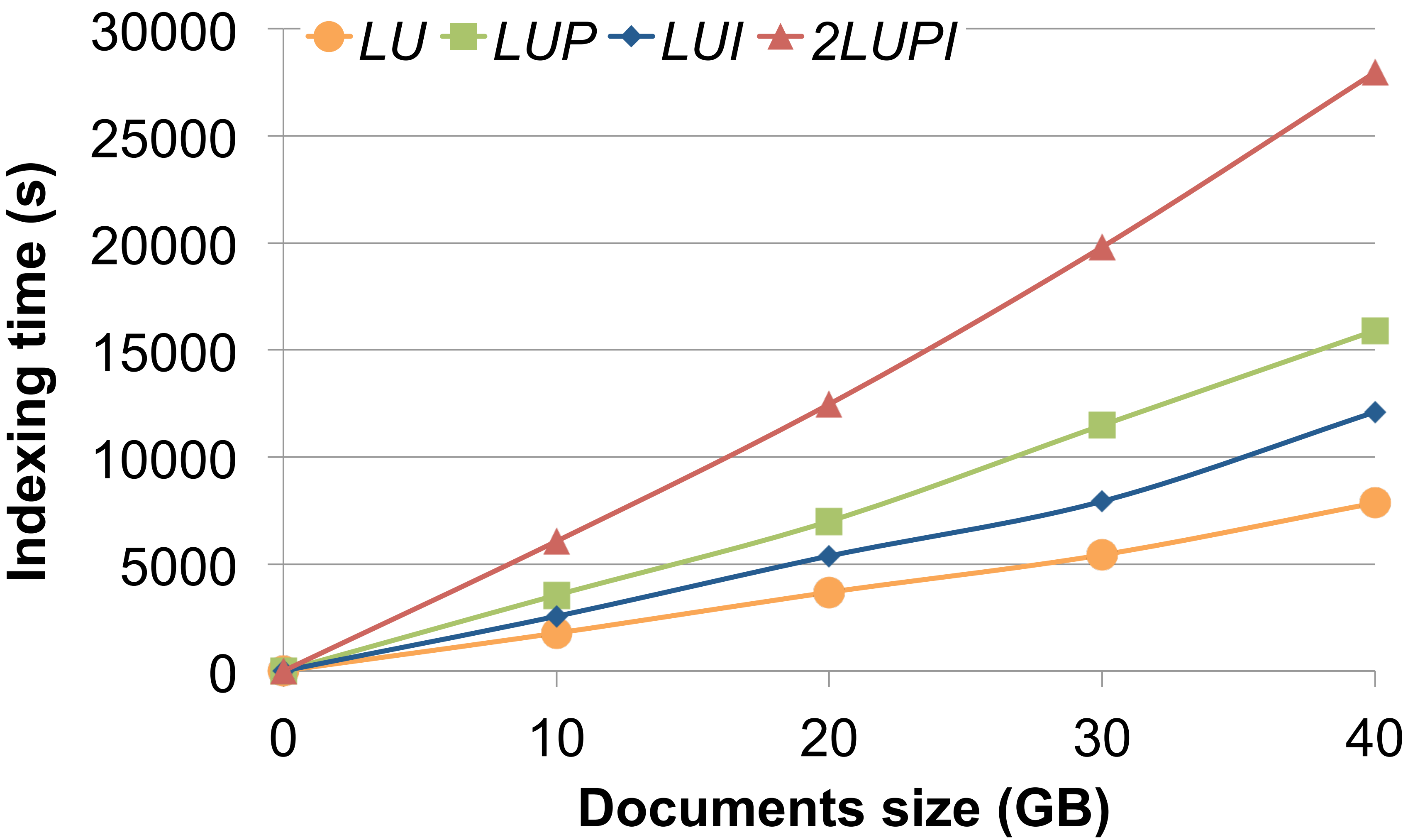
Look-up: Structural join over IDs associated to each query node



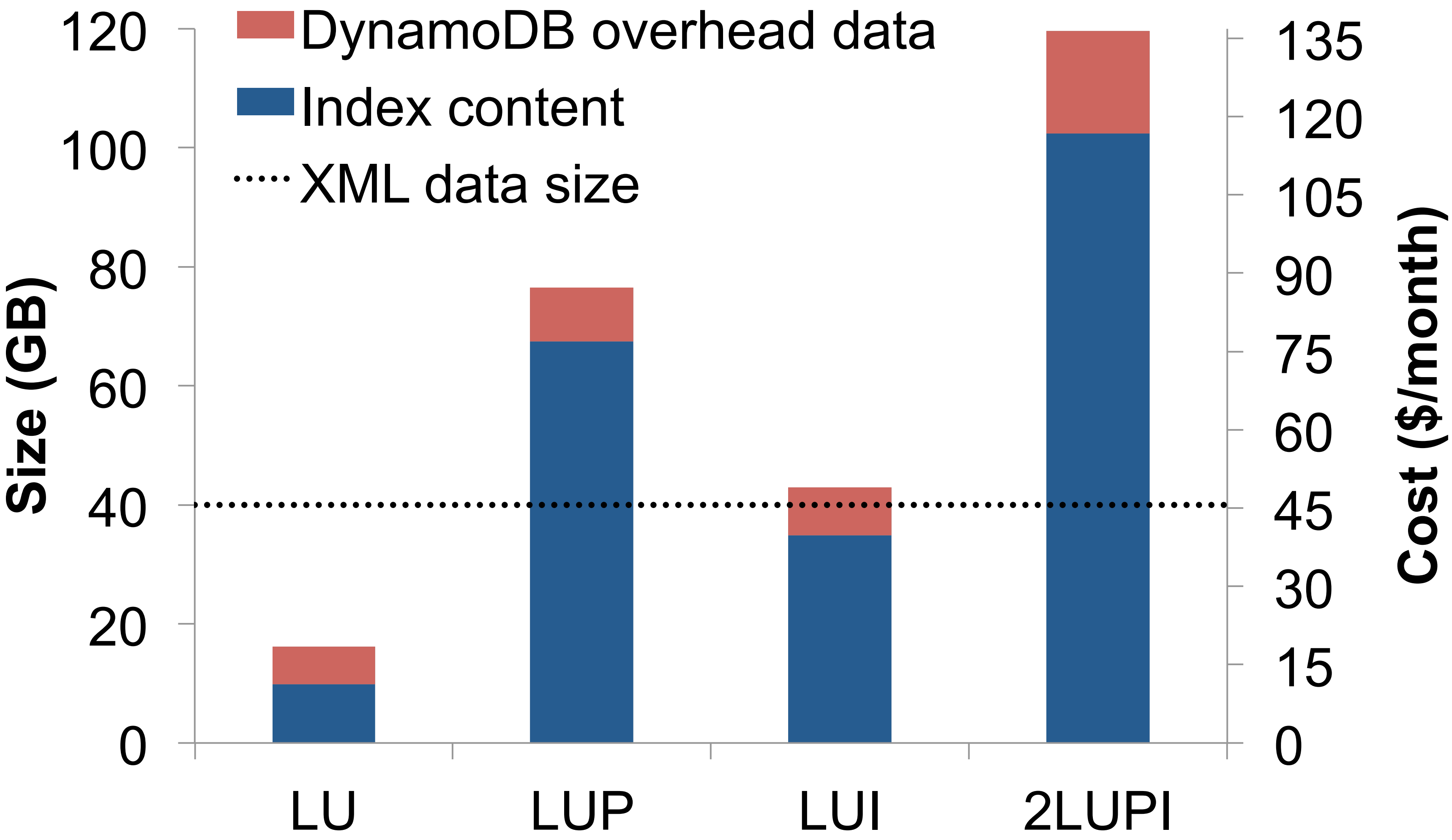
Experimental evaluation

- **AMADA** implemented in Java
  - Amazon Web Services SDK for Java
  - In-house XML query processor
- Two types of EC2 instances
  - Large (L), 7.5 GB of RAM memory and 2 virtual cores
  - Extra large (XL), 15 GB of RAM memory and 4 virtual cores
- 40GB XML documents
- Numbers from 2013

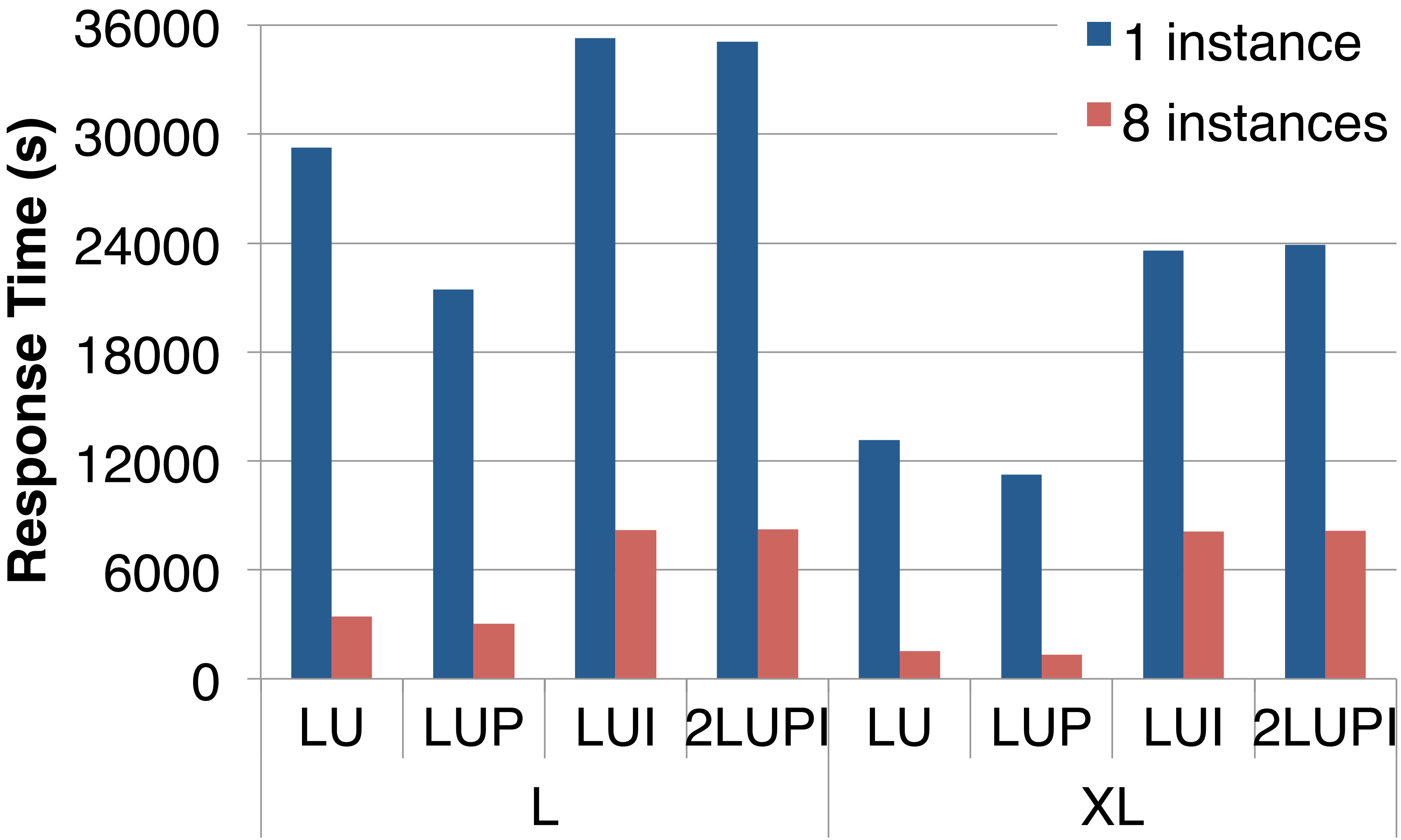
Index creation (8 XL, 20000 documents)



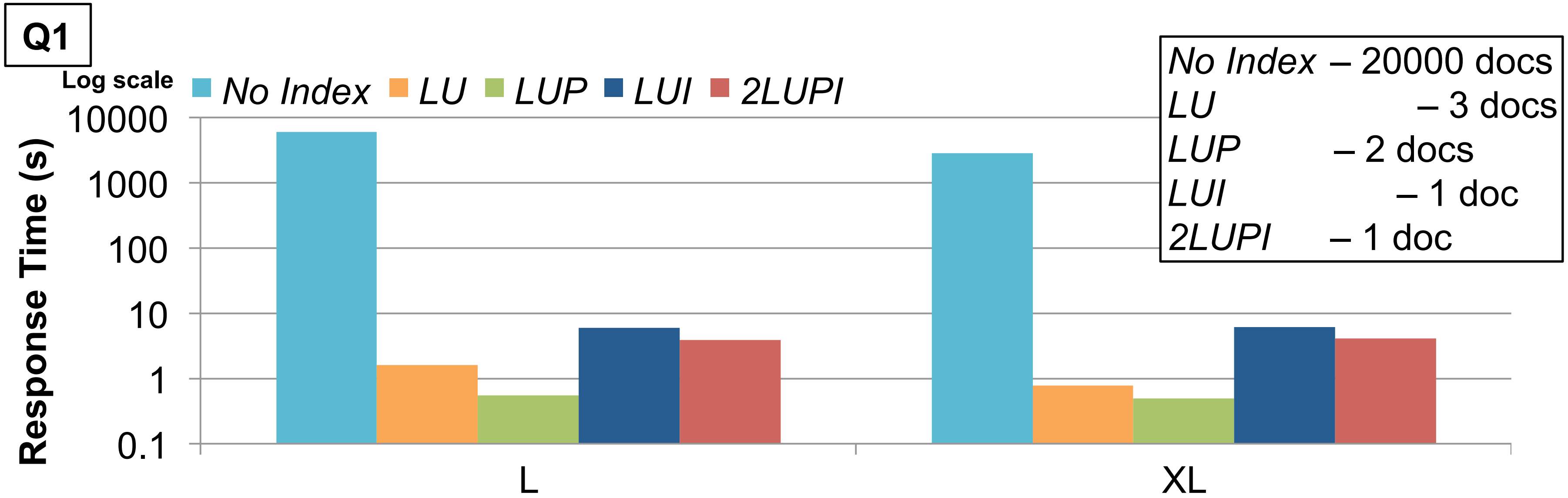
Index storage (20000 documents)



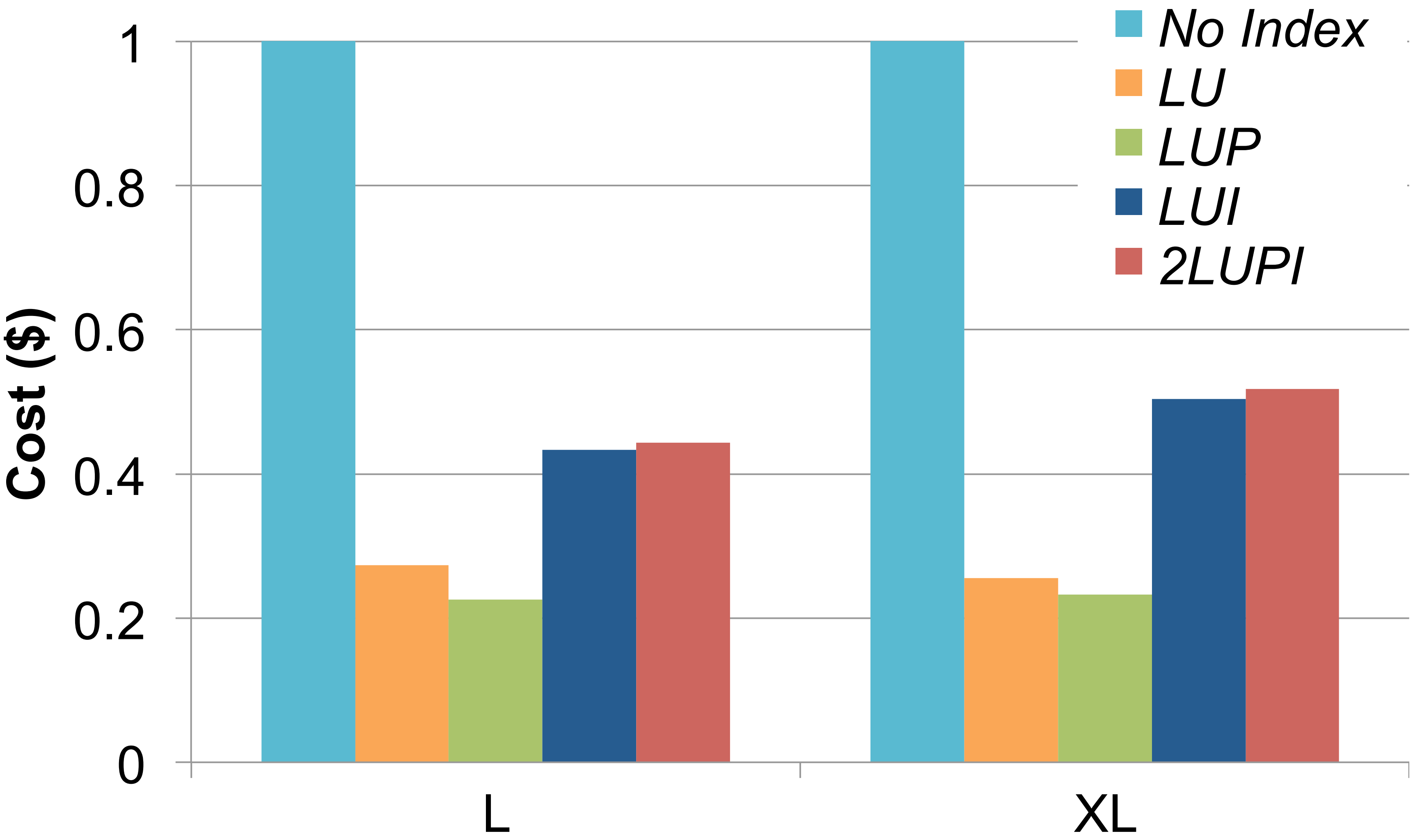
Query answering (eight runs)



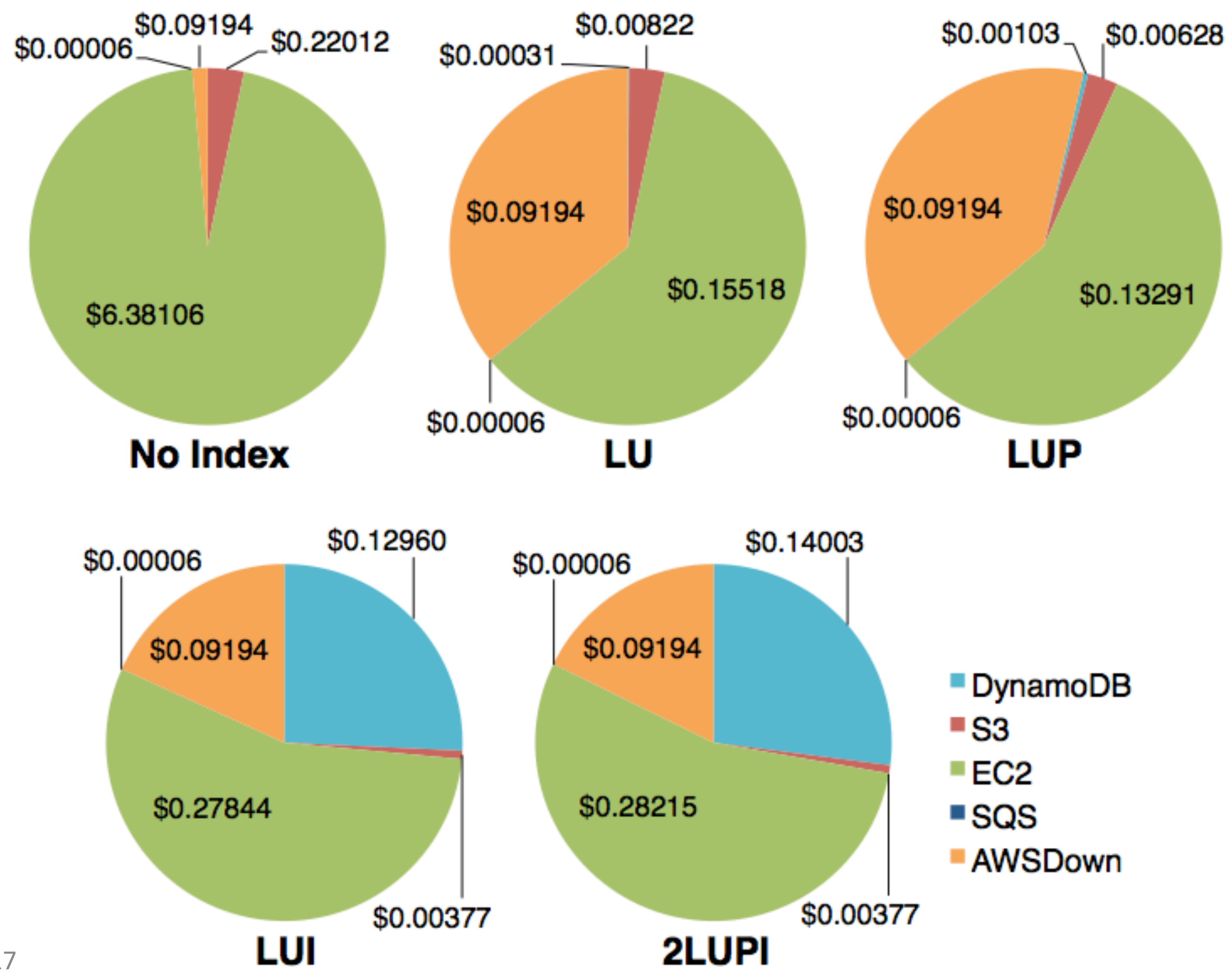
Query answering time



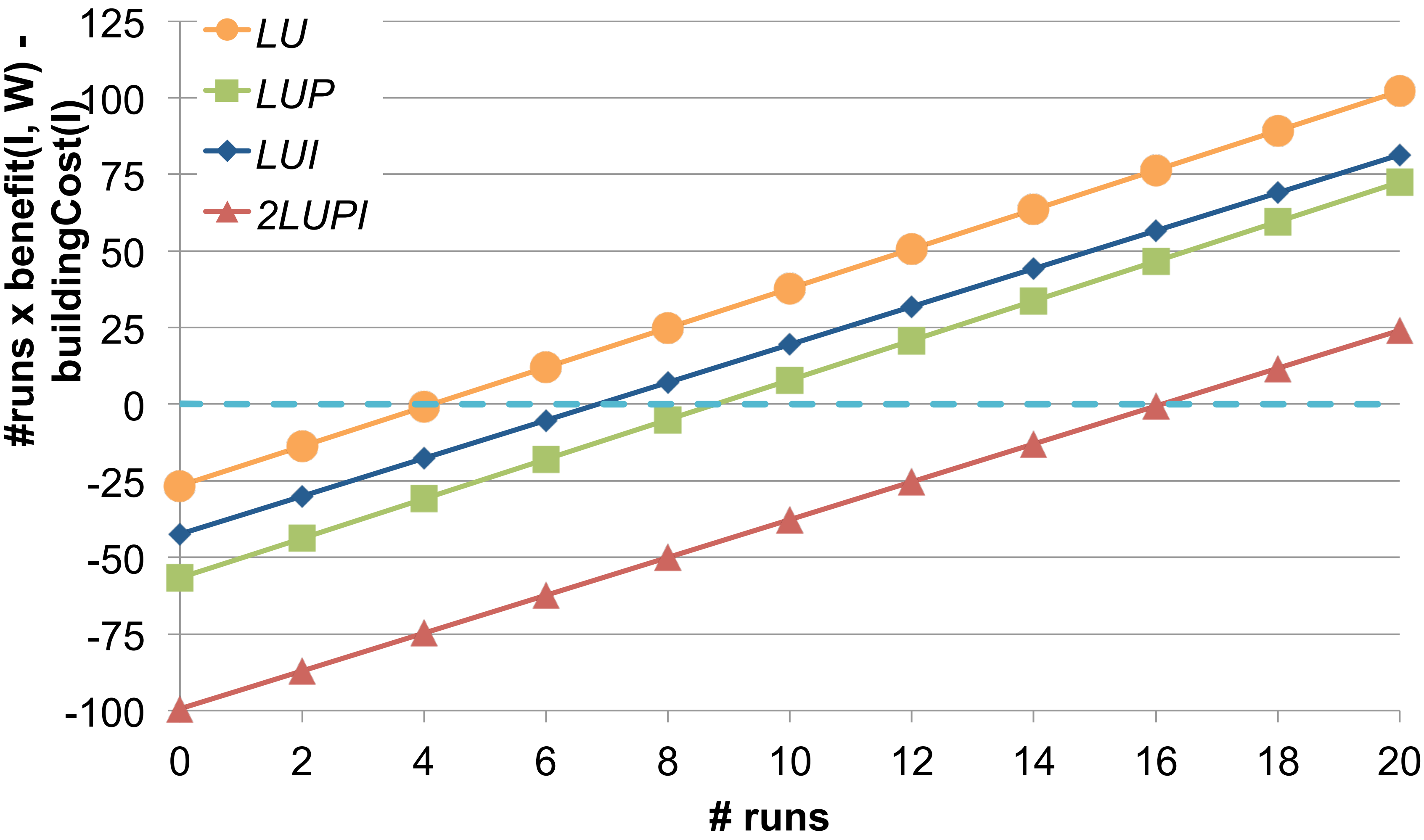
Query answering cost



Query answering cost detail (XL)



Index cost amortization



Conclusion

- Large-scale data storage platforms provide
  1. back-end for distributed storage (e.g. distributed file system, or cloud file store)
  2. [small-granularity, fast-access data store]
  3. [default computation model, e.g., MapReduce]
- To get a platform, still need to add/chose:
  1. Data model; query language
  2. Way to split, store [, index] the data
  3. Compiling the language to the computing model
  4. [Algebraic optimization]
  5. Consistency model
- Very fertile playing ground

References

- [CCH+16] **Reuse-based Optimization for PigLatin**. Jesus Camacho-Rodriguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, Soudip Roy-Chowdhury. CIKM, 2016
- [CCM13] Web data indexing in the cloud: efficiency and cost reductions. Jesus Camacho-Rodriguez, Dario Colazzo, Ioana Manolescu. EDBT, 2013
- [CDG+06] **Bigtable: A Distributed Storage System for Structured Data**. Fay Chang, Jeffrey Dean, Sanjay Chemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber. OSDI, 2006
- [CDE+12] **Spanner: Google's Globally-Distributed Database**. James Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes et al. OSDI, 2012
- [SVS+13] **F1: A Distributed SQL Database That Scales**. Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey et al. PVLDB, 2013