CMSC 724: Database Management Systems
Data Streams and Dataflow Engines

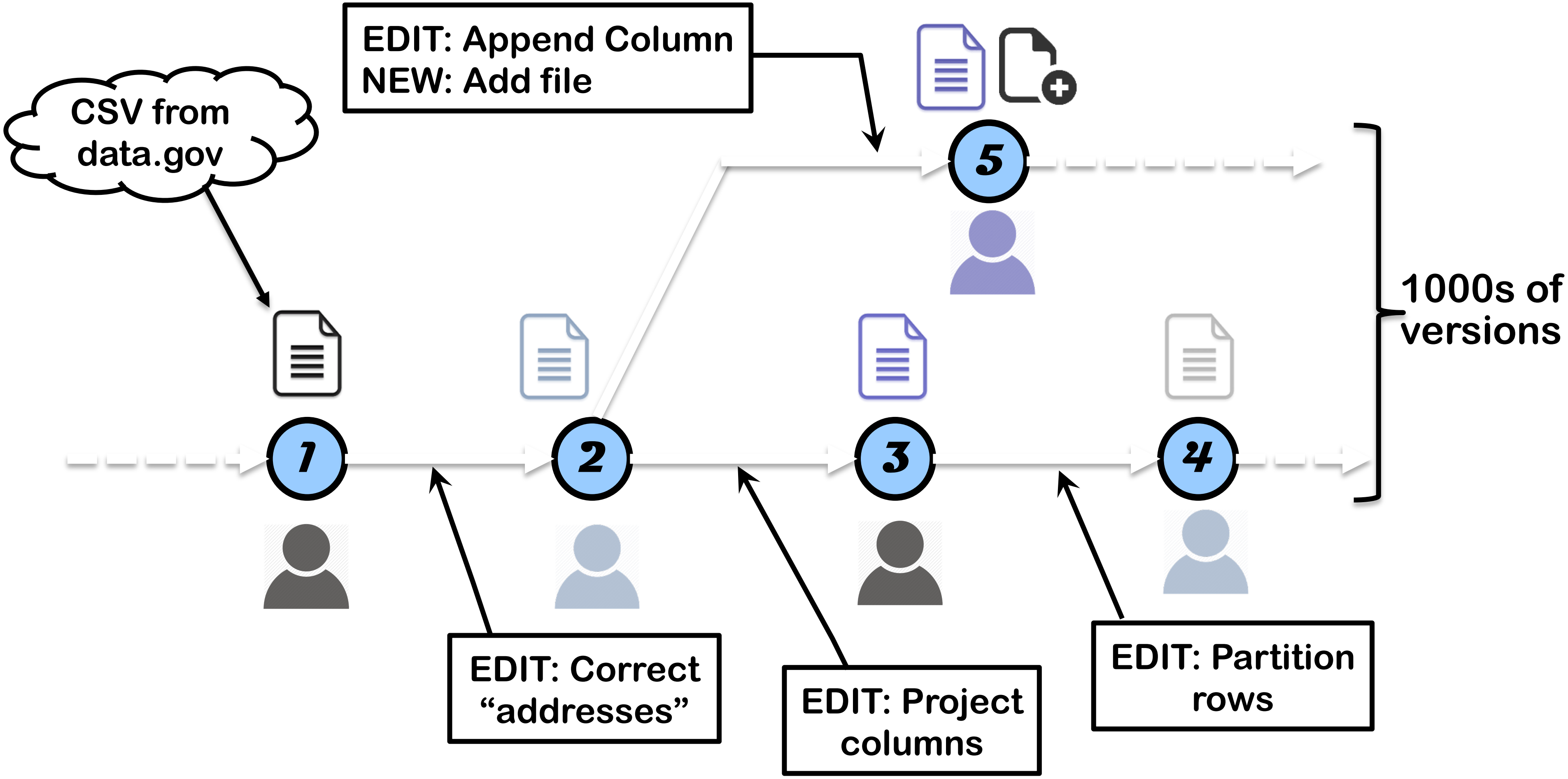
Instructor: Amol Deshpande
amol@cs.umd.edu

Outline

- DataHub: Overview
- OrpheusDB
- TardisDB
- Forkbase

Collaborative Data Science

- Widespread use of “data science” in many many domains



CSV from data.gov

EDIT: Append Column
NEW: Add file

EDIT: Correct “addresses”

EDIT: Project columns

EDIT: Partition rows

1000s of versions

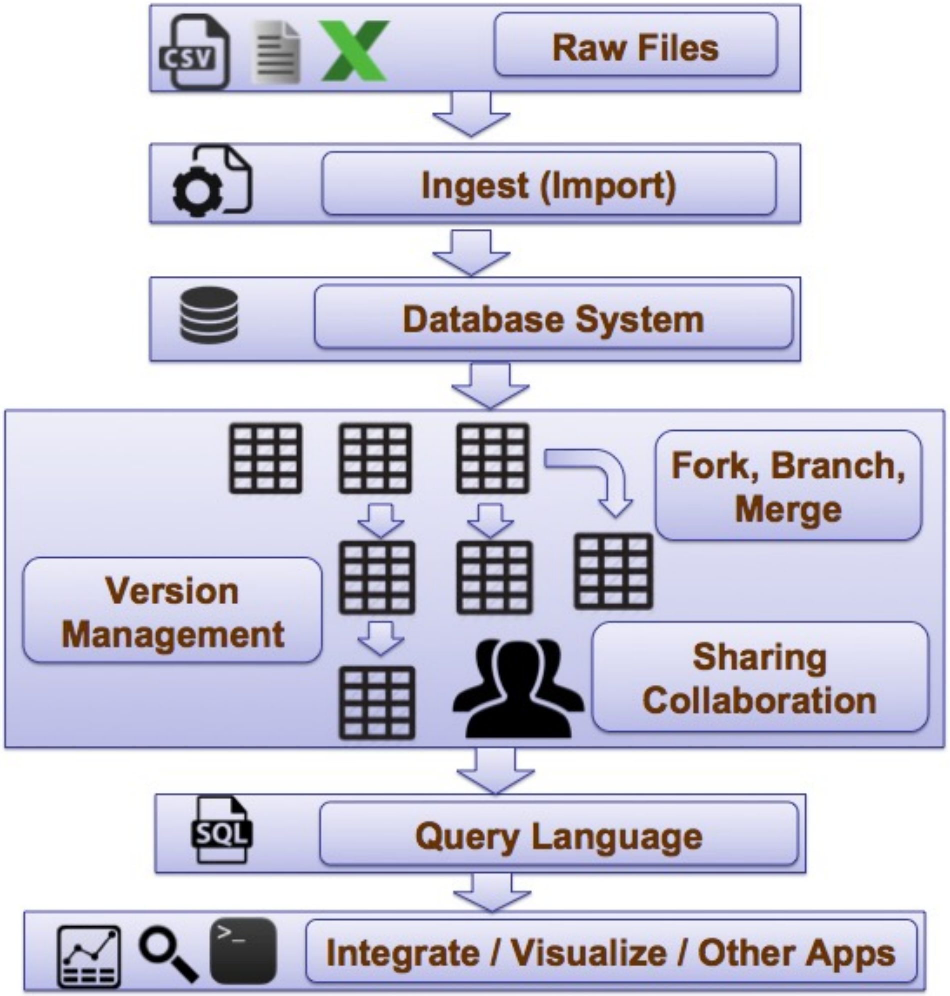
A typical data analysis workflow

Collaborative Data Science

- Widespread use of “data science” in many many domains
- Increasingly the “pain point” is managing the *process,* especially during collaborative analysis
  - Many private copies of the datasets ➔ Massive redundancy
  - No easy way to keep track of dependencies between datasets
  - Manual intervention needed for resolving conflicts
  - No efficient organization or management of datasets
  - No way to analyze/compare/query versions of a dataset
- Ad hoc data management systems (e.g., Dropbox) used
  - Much of the data is unstructured so typically can’t use DBs
  - The process of data science itself is quite ad hoc and exploratory
  - Scientists/researchers/analysts are pretty much on their own

DataHub: A Collaborative Data Science Platform

The one-stop solution for collaborative data science and dataset version management



http://data-hub.org

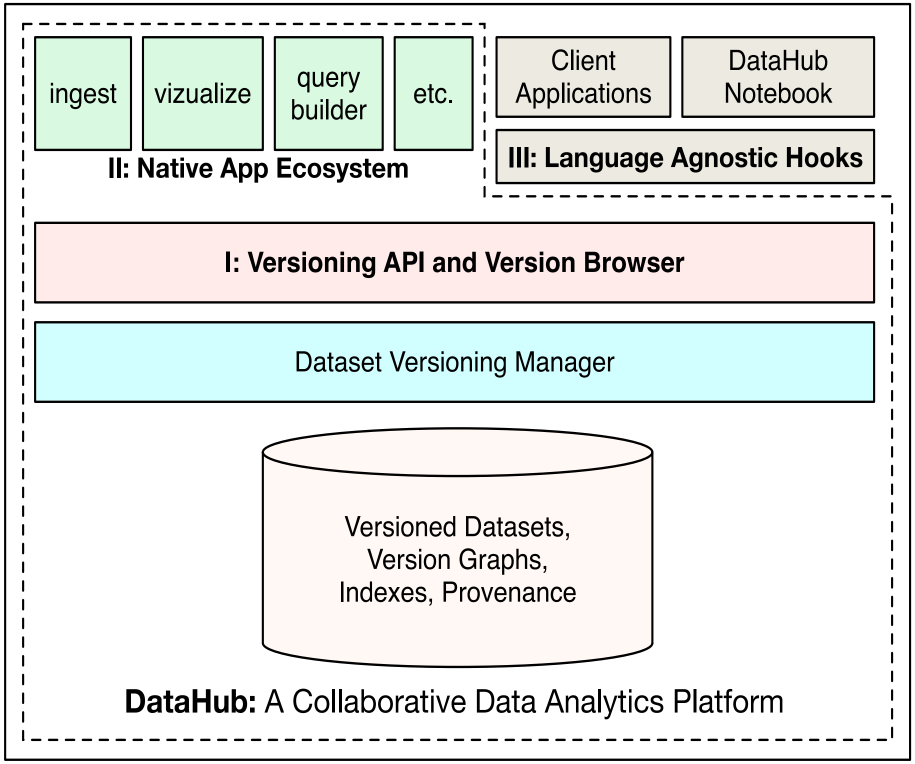
Work being done in collaboration with Sam Madden (MIT) and Aditya Parameswaran (UIUC)

DataHub: A Collaborative Data Science Platform

- a dataset management system – import, search, query, analyze a large number of (public) datasets

- a dataset version control system – branch, update, merge, transform large structured or unstructured datasets

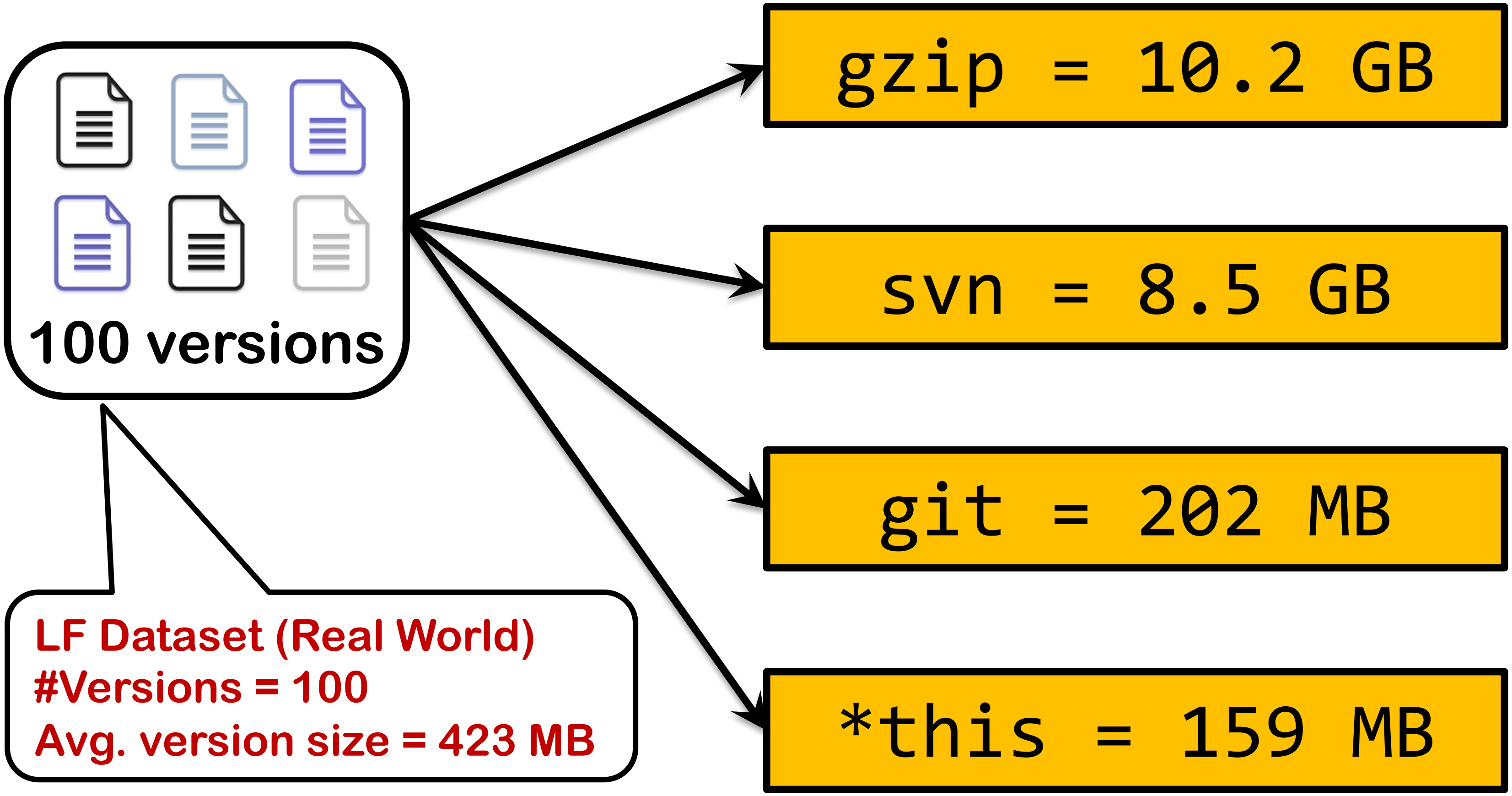
- an app ecosystem and hooks for external applications (Matlab, R, iPython Notebook, etc)



DataHub Architecture

Can we use Version Control Systems (e.g., Git)?

- ✗ **No, because they typically use fairly simple algorithms and are optimized to work for code-like data**



100 versions

gzip = 10.2 GB

svn = 8.5 GB

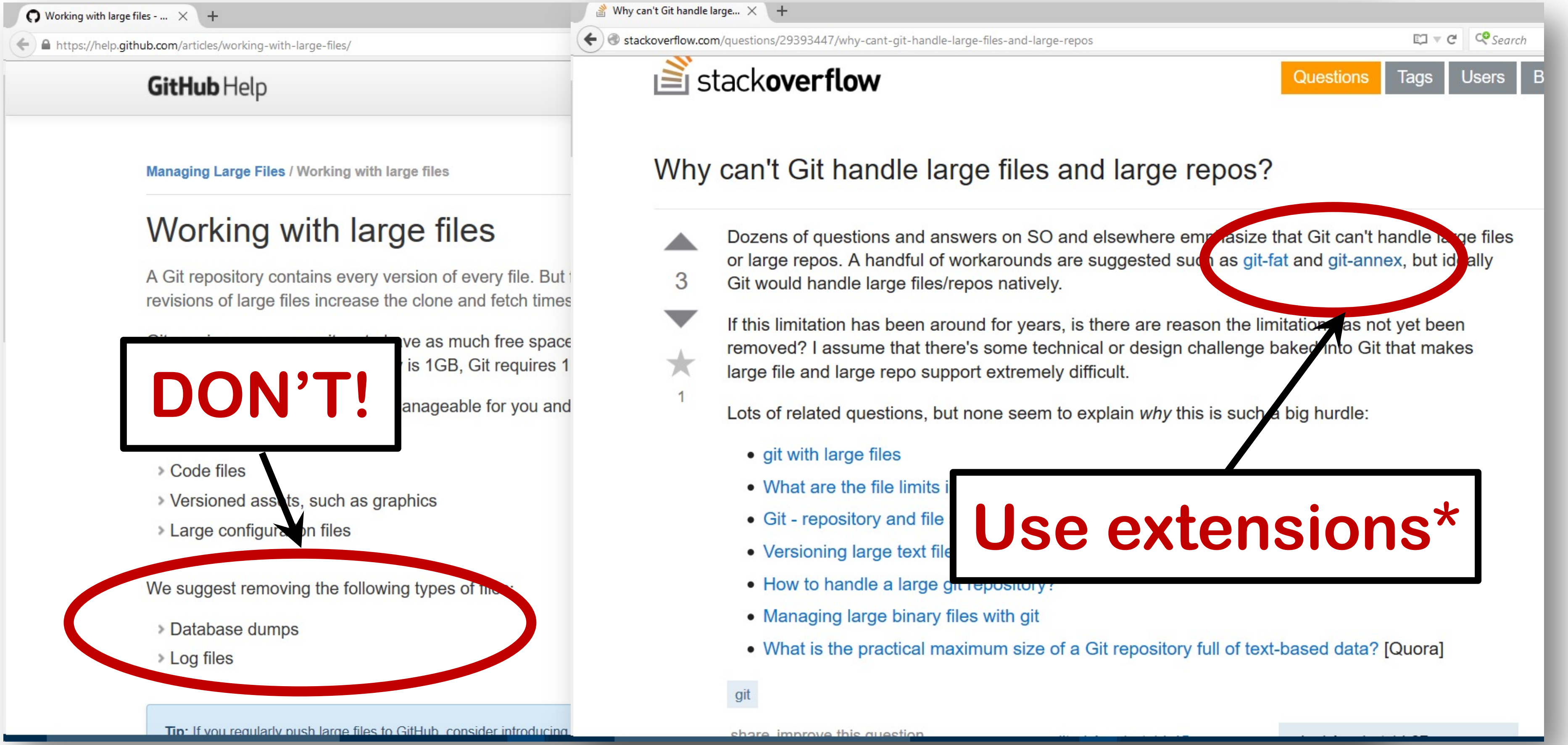
git = 202 MB

*this = 159 MB

LF Dataset (Real World)
#Versions = 100
Avg. version size = 423 MB

Can we use Version Control Systems (e.g., Git)?

- ✗ No, because they typically use fairly simple algorithms and are optimized to work for code-like data
- ✗ Git ends up using large amounts of RAM for large files



Managing Large Files / Working with large files

Working with large files

A Git repository contains every version of every file. But

revisions of large files increase the clone and fetch times

ve as much free space

is 1GB, Git requires 1

anageable for you and

- Code files
- Versioned assets, such as graphics
- Large configuration files

We suggest removing the following types of files

- Database dumps
- Log files

Tip: If you regularly push large files to GitHub, consider introducing

DON'T!

Why can't Git handle large files and large repos?

Dozens of questions and answers on SO and elsewhere emphasize that Git can't handle large files or large repos. A handful of workarounds are suggested such as git-fat and git-annex, but ideally Git would handle large files/repos natively.

If this limitation has been around for years, is there are reason the limitation has not yet been removed? I assume that there's some technical or design challenge baked into Git that makes large file and large repo support extremely difficult.

Lots of related questions, but none seem to explain *why* this is such a big hurdle:

- git with large files
- What are the file limits i
- Git - repository and file
- Versioning large text file
- How to handle a large git repository?
- Managing large binary files with git
- What is the practical maximum size of a Git repository full of text-based data? [Quora]

Use extensions*

Can we use Version Control Systems (e.g., Git)?

- ✗ **No, because they typically use fairly simple algorithms and are optimized to work for code-like data**
- ✗ **Git ends up using large amounts of RAM for large files**
- ✗ **Querying and retrieval functionalities are primitive, and revolve around single version and metadata retrieval**
- ✗ **No way to specify queries like:**
  - *identify all datasets derived of dataset A that satisfy property P*
  - *identify all predecessor versions of version A that differ from it by a large number of records*
  - *rank a set of versions according to a scoring function*
  - *find the version where the result of an aggregate query is above a threshold*
  - *find parent records of all records in version A that satisfy certain property*

DSVC Data Model [CIDR 2015]

- Schema-later Data Representation
  - Base model is that of key-value pairs
- Version Graph
  - Information about how versions are created and relate to each other
- Versioning API
  - create, branch, merge, commit, rollback, checkout
  - “hooks” to run scripts before/after/during “commits”
- Transaction mode (similar to a typical server-based DBMS), vs local mode (similar to “git”)
  - Former is not straightforward to do

Query Language[CIDR 2015]

- [[Note: A more comprehensive proposal in a later paper]]
- Supports queries on the datasets within a version, as well as queries about the version graph
- Ability to mix those two as well

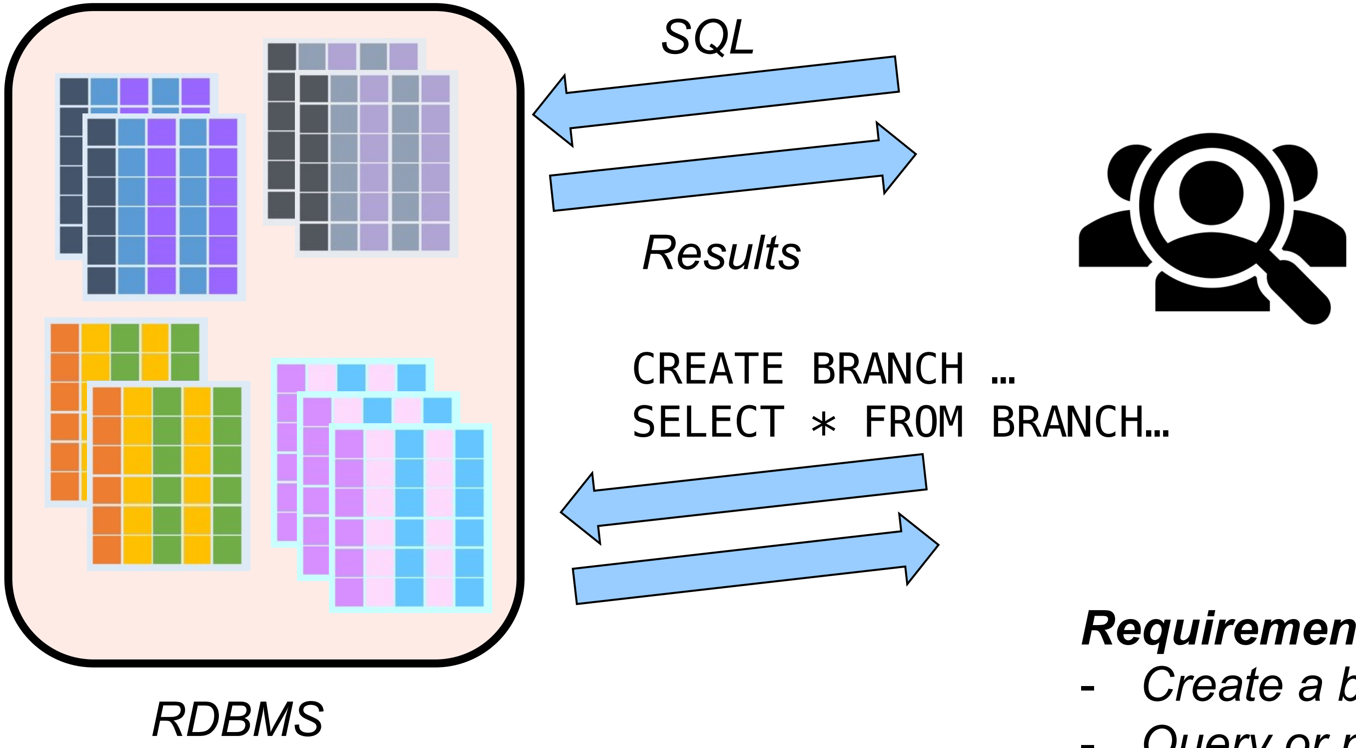
```
SELECT * FROM R(v124), R(v135)
WHERE R(v124).id = R(v135).id
```

```
SELECT * FROM S(SELECT MIN(VR1.VNUM) FROM
VERSIONS(R) VR1, VERSIONS(R) VR2
WHERE DISTANCE(R,VR1.VNUM,VR2.VNUM)=1
AND DIFF_RECS(R,VR1.VNUM,VR2.VNUM)>100)
```

Dataset Versioning and Compression

- Many different ”overlap” structures
  - Dependent heavily on the type of data, and the types of modifications on them
- Varying computational environments
  - Distributed vs centralized
  - ”Check out” or “in situ” processing
- Different ”retrieval” requirements
  - Full versions vs small portions of versions
  - Analysis across one version or many versions
- Need support for ACID transactions and rich querying
  - For operation databases, or data warehouses

Scenario 1: Relational Database



```
CREATE BRANCH …
SELECT * FROM BRANCH…
```

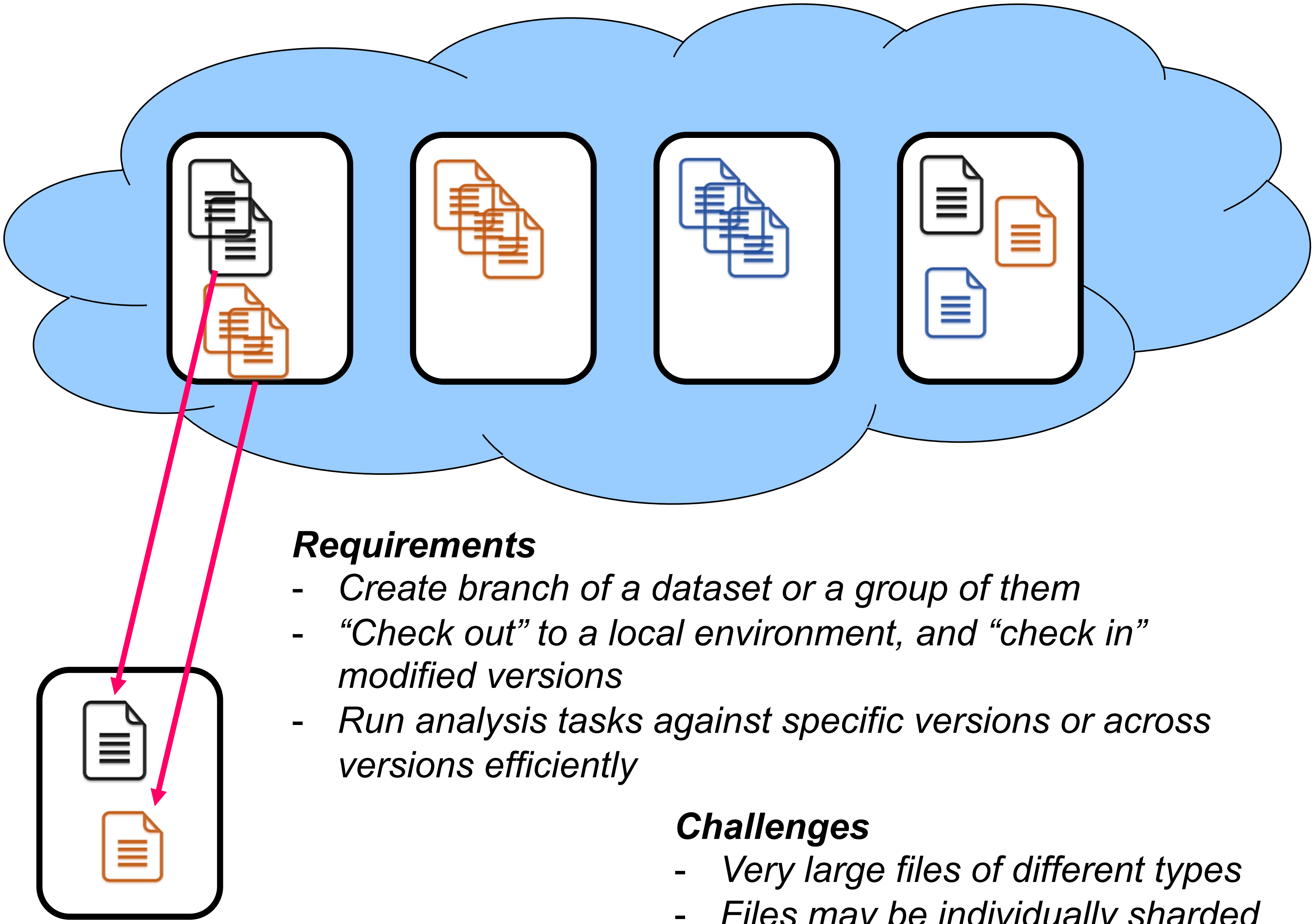
Requirements

- *Create a branch of the database*
- *Query or modify specific branches*
- *Merge branches*
- *…*

Challenges

- *Not feasible to "check out" locally – need to support "in situ" processing*
- *Need to maintain many branches simultaneously in a single server*
- *Need to redesign internal data structures, transaction engines, etc.*

Scenario 2: Files in Data Lakes



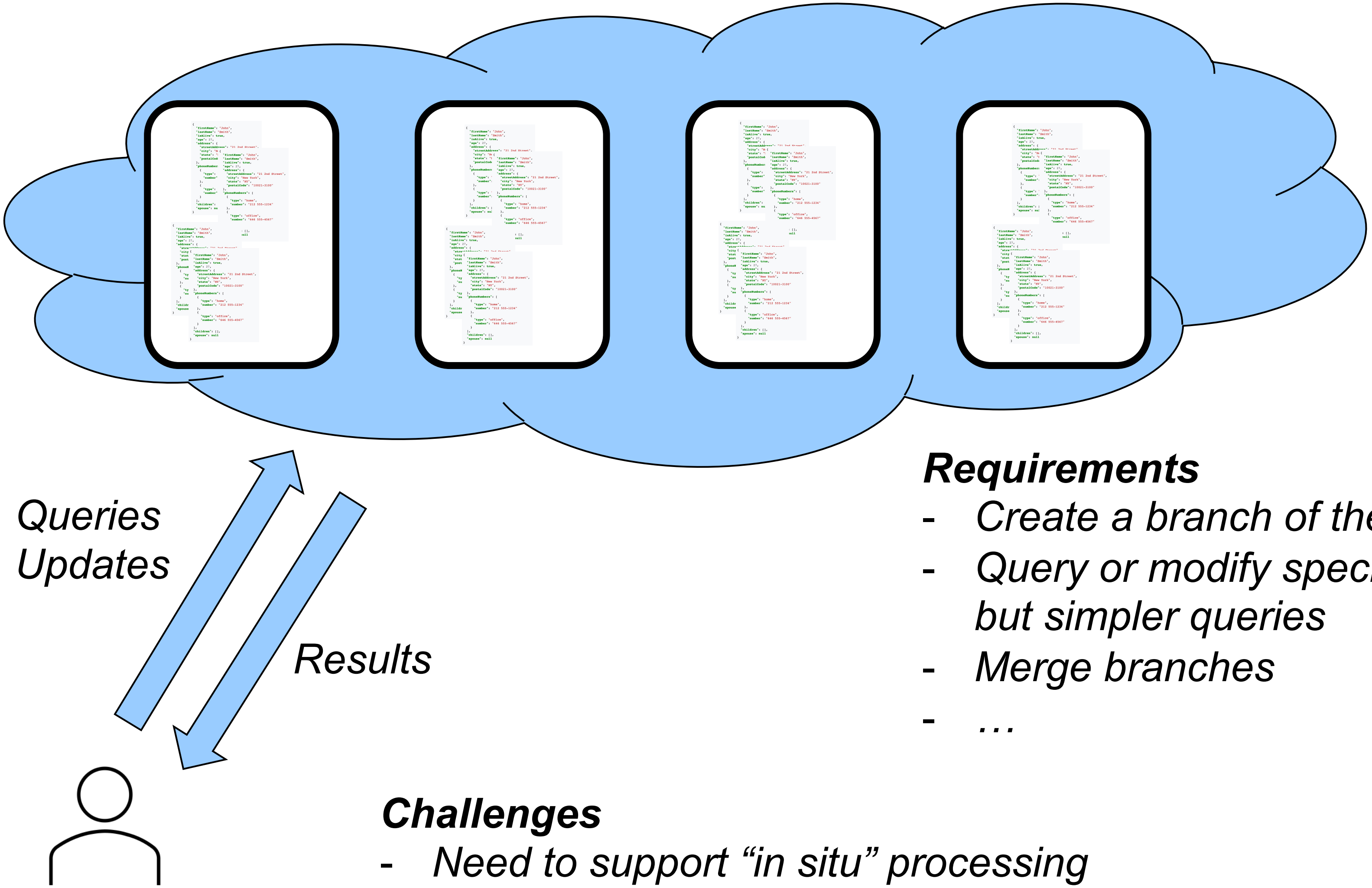
Requirements

- *Create branch of a dataset or a group of them*
- *“Check out” to a local environment, and “check in” modified versions*
- *Run analysis tasks against specific versions or across versions efficiently*

Challenges

- *Very large files of different types*
- *Files may be individually sharded*

Scenario 3: Distributed Document Store



Requirements

- *Create a branch of the database*
- *Query or modify specific branches, but simpler queries*
- *Merge branches*
- *…*

Challenges

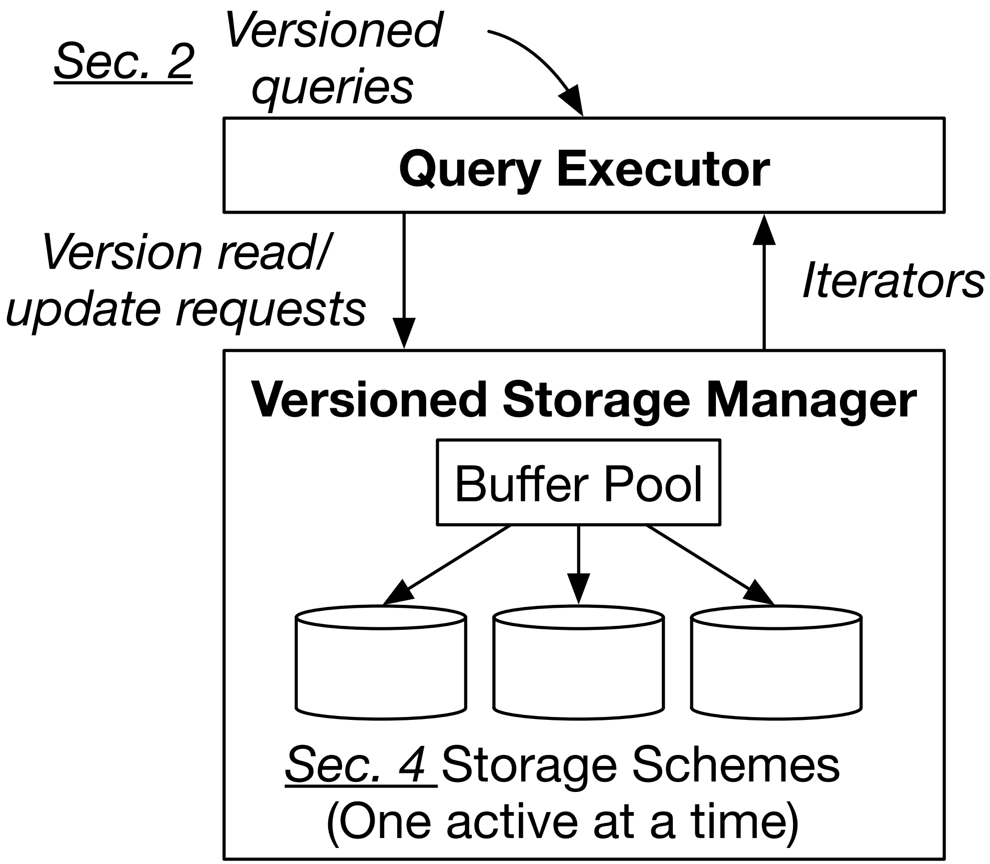
- *Need to support "in situ" processing*
- *Must minimize the number of queries to the backend store*
- *Need to support "key-based" retrieval*
- *Documents typically large (in MBs), with small changes*

Scenario 1: Relational Databases

Decibel [VLDB'18]

Modified the "storage manager" for MIT SimpleDB RDBMS

Supports branching and merging, and queries across versions (e.g., diffs)



Storage Strategies

Key Observation: *Differences across versions/branches are presence or absence of individual tuples (or tuple attributes)*

Can be captured as a binary "membership" matrix

Branches

Tuples

	B1	**B2**	**B3**	**…**	**…**
t1	1	0	0	0	0
t2	0	1	0	0	0
t3	0	0	0	0	1
…	0	0	0	1	0
…	1	0	0	0	0
	0	1	0	0	0
	0	1	0	0	0
	0	1	0	0	0
	0	0	0	0	1
	1	0	0	0	0

Typically: tall and narrow
 # branches << # tuples

Compressing binary matrixes is a well-studied problem (NP-Hard in general)

However, we need to support:

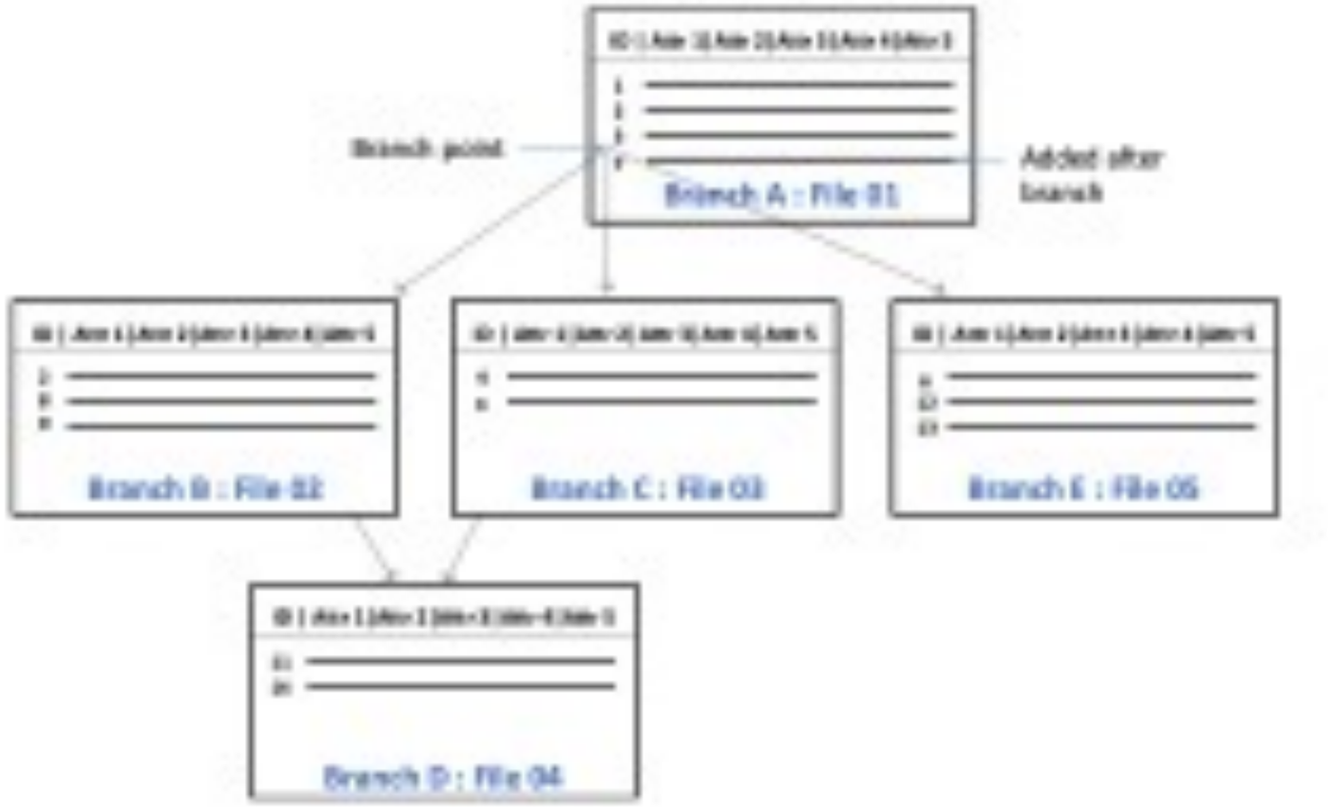
- *Efficient updates*
- *Retrieval of one or more versions*
- *Queries on specific columns (branches)*
- *Queries across pairs or groups of versions*

Tuple-first Storage Strategies

Compressed bitmap per branch, vs per tuple

- *Also need to consider how the bitmaps will be compressed (e.g., run-length encoding) and how they will be mapped to memory block*
- *Commit operations easier for bitmap-per-branch, but tuple inserts faster in bitmap-per-tuple*
- *Queries across branches, including ”merges”, can exploit bitmap operations*

Version-first Storage Strategies



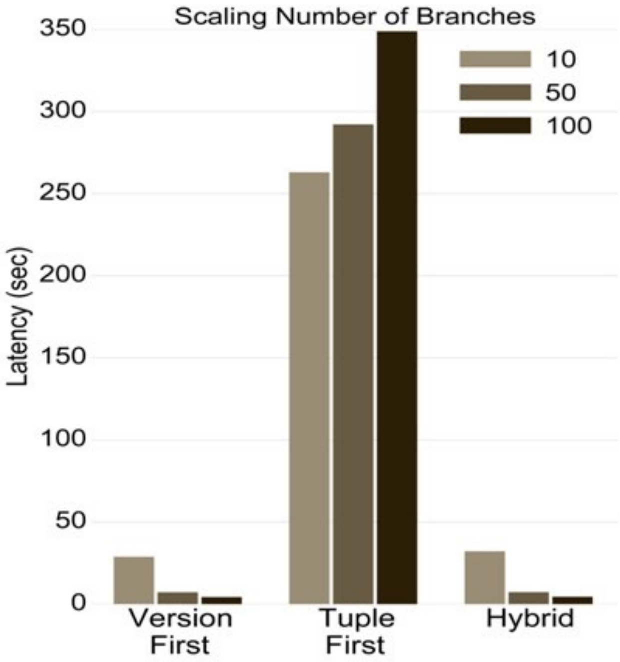
- *Use “deltas” across versions (i.e., tuple differences)*
- *Better when changes across versions are small*
- *Performance of queries across versions poor*

Some Experimental Results

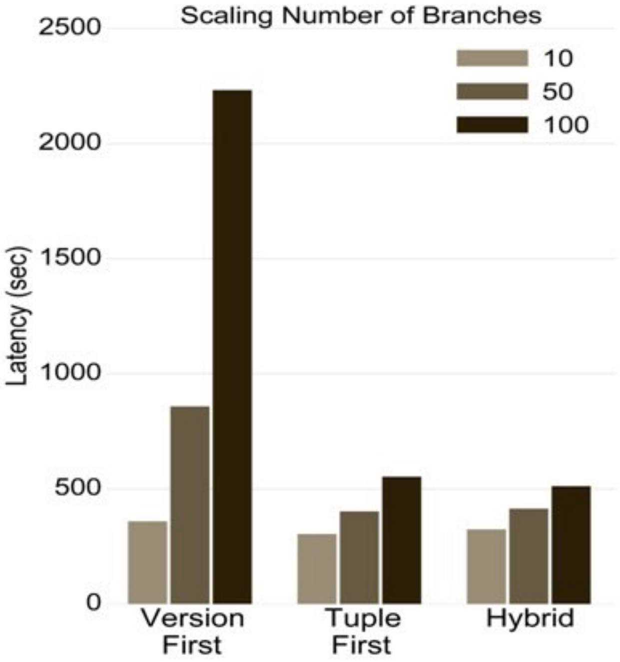
	Data Size (GB)	Load Time (sec)	Repo Size (MB)	Checkout Avg. (ms)	Commit Avg. (ms)
git	1	615	375	2100	5400
Decibel	1	7	1002	4	5
git	2	16 204	5620	242 000	31 400
Decibel	2	12	2011	8	6

Comparing git and Decibel (Hybrid)

Single-version Scan on a Flat Version Graph



Multi-version Scan on a Deep Version Graph



Open Research Questions

- Handling schema changes
  - Would like to version schemas along with data
  - More complex compression problems
- Better compression algorithms for more efficient handling of large numbers of versions
- Handling deletes and merges more cleanly
  - Especially conflicts during merges
- Interactions with other database components
  - Concurrency, Recovery, Query Processing and Optimization, etc.

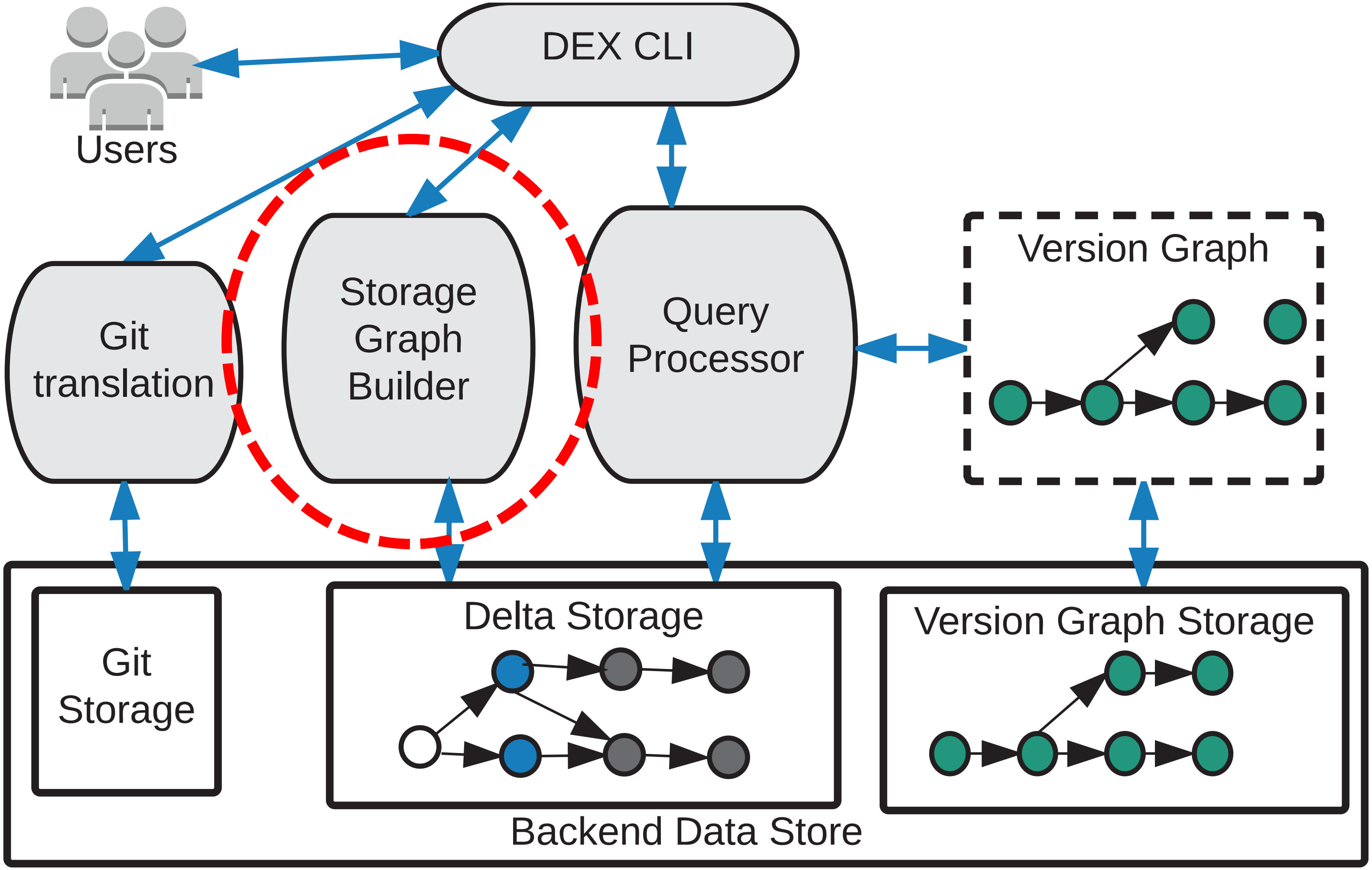
Scenario 2: Files in Data Lakes

DEX: Delta-oriented EXecution Engine

[VLDB'15, VLDB'16, SIGMOD'17]

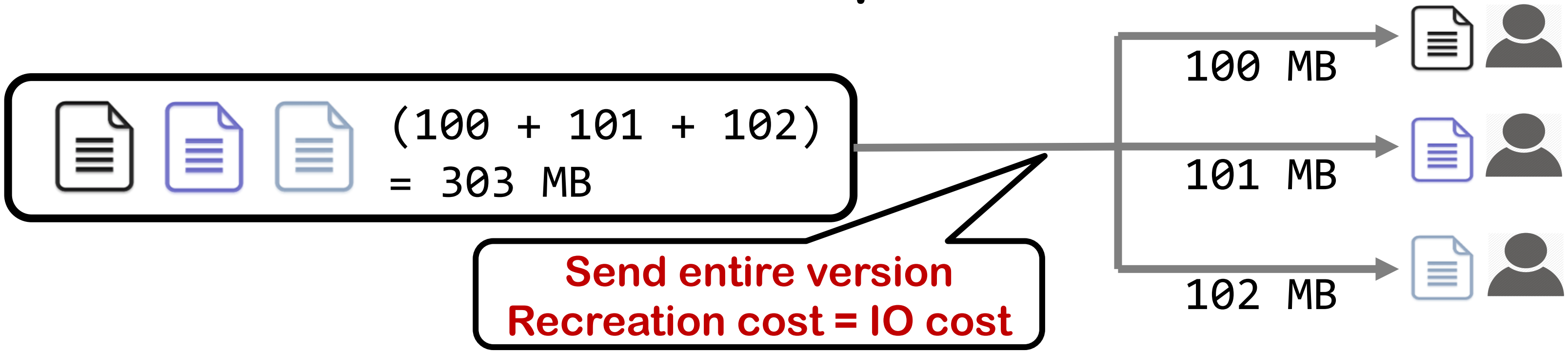
Built as a "git" extension

Supports standard checkout/commit etc., operations against files



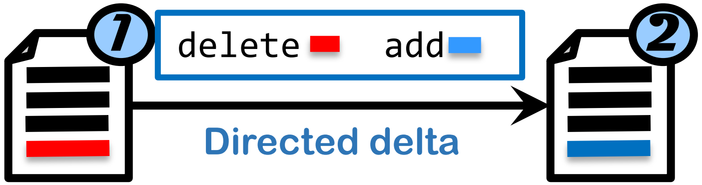
Storage cost is the space required to store a set of versions

Recreation cost is the time* required to access a version

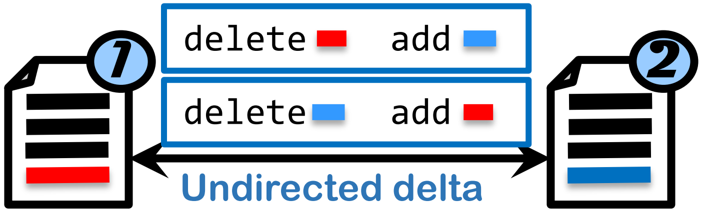


Send entire version
Recreation cost = IO cost

A **delta** between versions is a file which allows constructing one version given the other



Directed delta

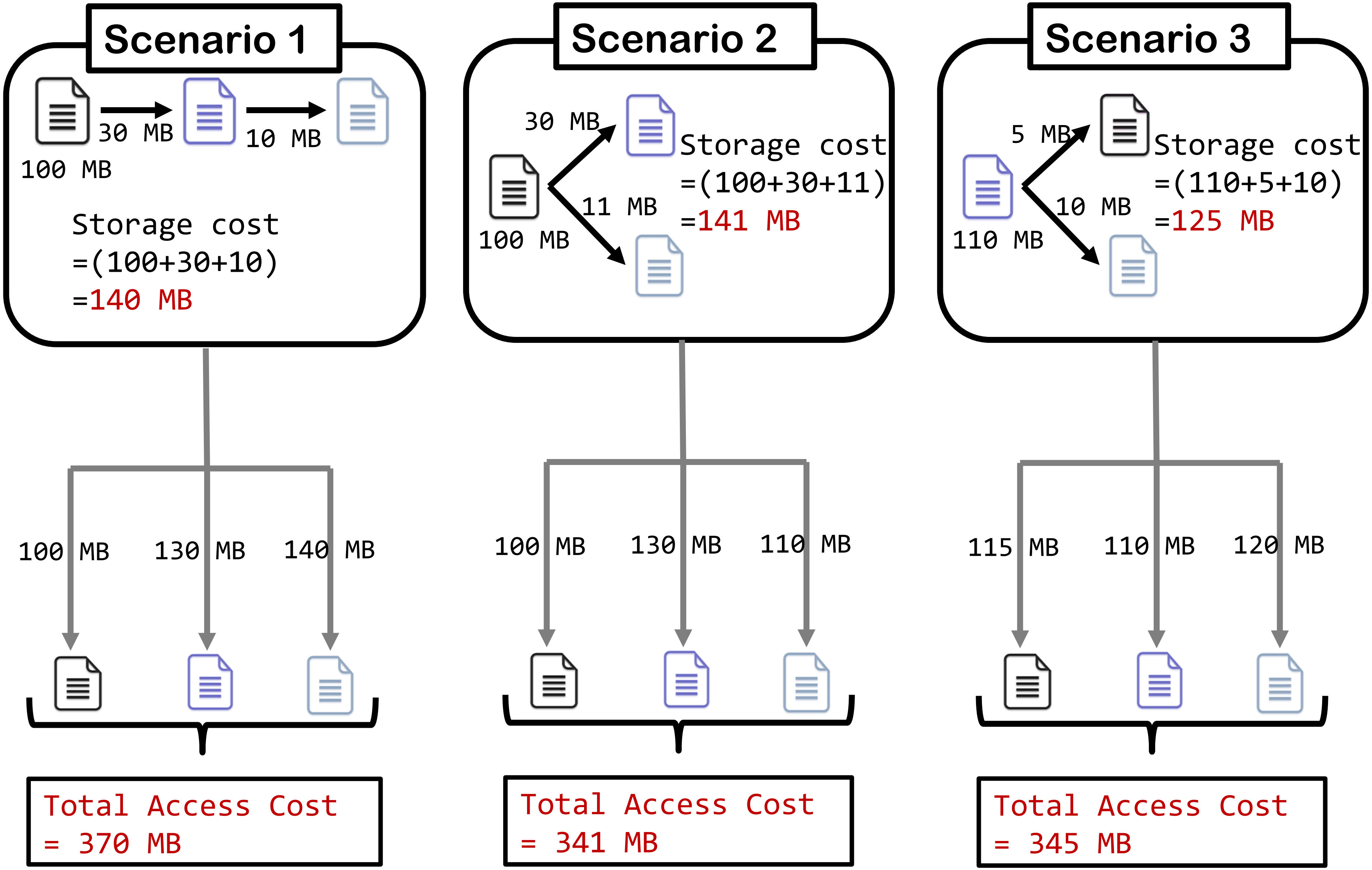


Undirected delta

Example: Unix diff, xdelta, XOR, etc.

A delta has its own **storage** cost and **recreation** cost, which, in general, are **independent** of each other

Storage-Recreation Tradeoff



Scenario 1

100 MB

30 MB

10 MB

Storage cost
=(100+30+10)
=140 MB

100 MB

130 MB

140 MB

Total Access Cost
= 370 MB

Scenario 2

30 MB

11 MB

100 MB

Storage cost
=(100+30+11)
=141 MB

100 MB

130 MB

110 MB

Total Access Cost
= 341 MB

Scenario 3

5 MB

10 MB

110 MB

Storage cost
=(110+5+10)
=125 MB

115 MB

110 MB

120 MB

Total Access Cost
= 345 MB

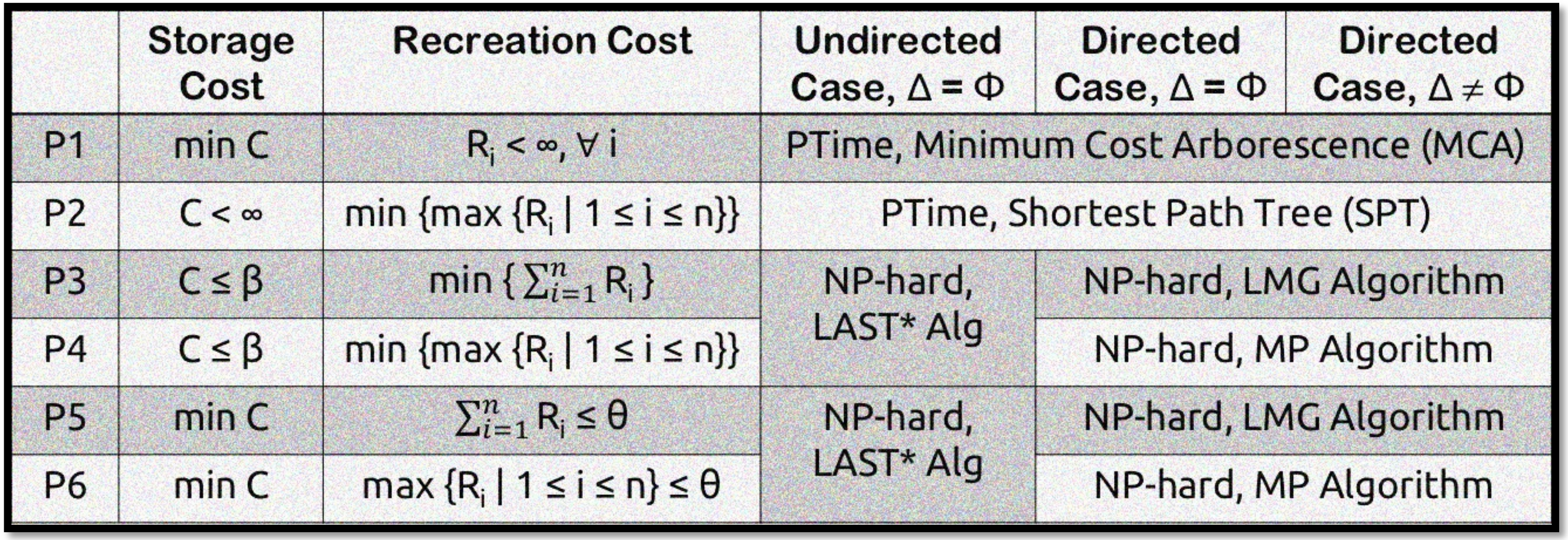
Storage-Recreation Tradeoff

Given

1) a set of versions
2) partial information about deltas between versions

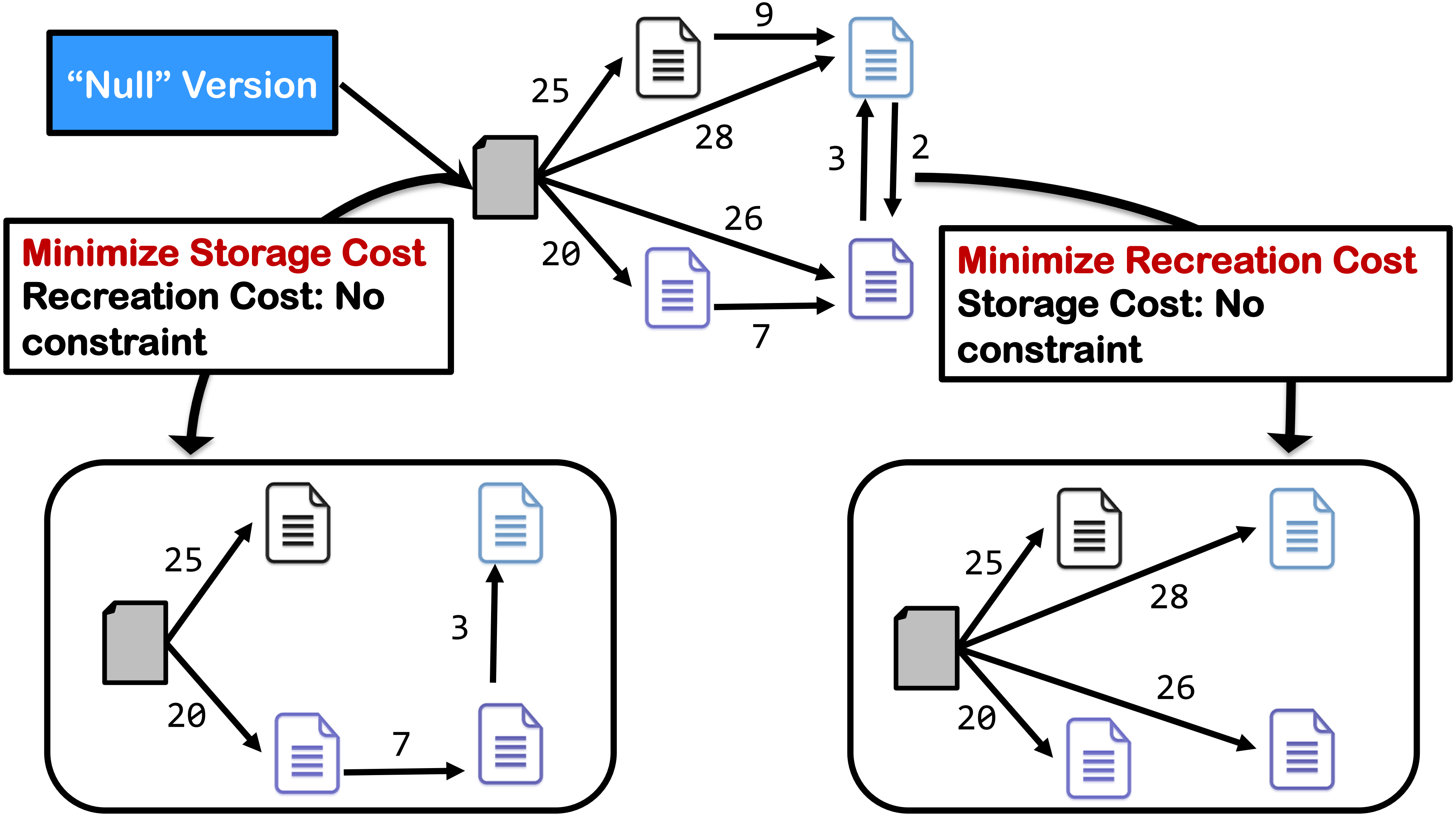
Find a Storage Solution that:

- minimizes total recreation cost given a storage budget, or
- minimizes max recreation cost given a storage budget



	Storage Cost	Recreation Cost	Undirected Case, $\Delta = \Phi$	Directed Case, $\Delta = \Phi$	Directed Case, $\Delta \neq \Phi$
P1	min C	$R_i < \infty, \forall i$	PTime, Minimum Cost Arborescence (MCA)		
P2	$C < \infty$	$\min \{\max \{R_i \mid 1 \leq i \leq n\}\}$	PTime, Shortest Path Tree (SPT)		
P3	$C \leq \beta$	$\min \{\sum_{i=1}^{n} R_i\}$	NP-hard, LAST* Alg	NP-hard, LMG Algorithm	
P4	$C \leq \beta$	$\min \{\max \{R_i \mid 1 \leq i \leq n\}\}$		NP-hard, MP Algorithm	
P5	min C	$\sum_{i=1}^{n} R_i \leq \theta$	NP-hard, LAST* Alg	NP-hard, LMG Algorithm	
P6	min C	$\max \{R_i \mid 1 \leq i \leq n\} \leq \theta$		NP-hard, MP Algorithm	

Baselines



Minimize Storage Cost
Recreation Cost: No constraint

Minimize Recreation Cost
Storage Cost: No constraint

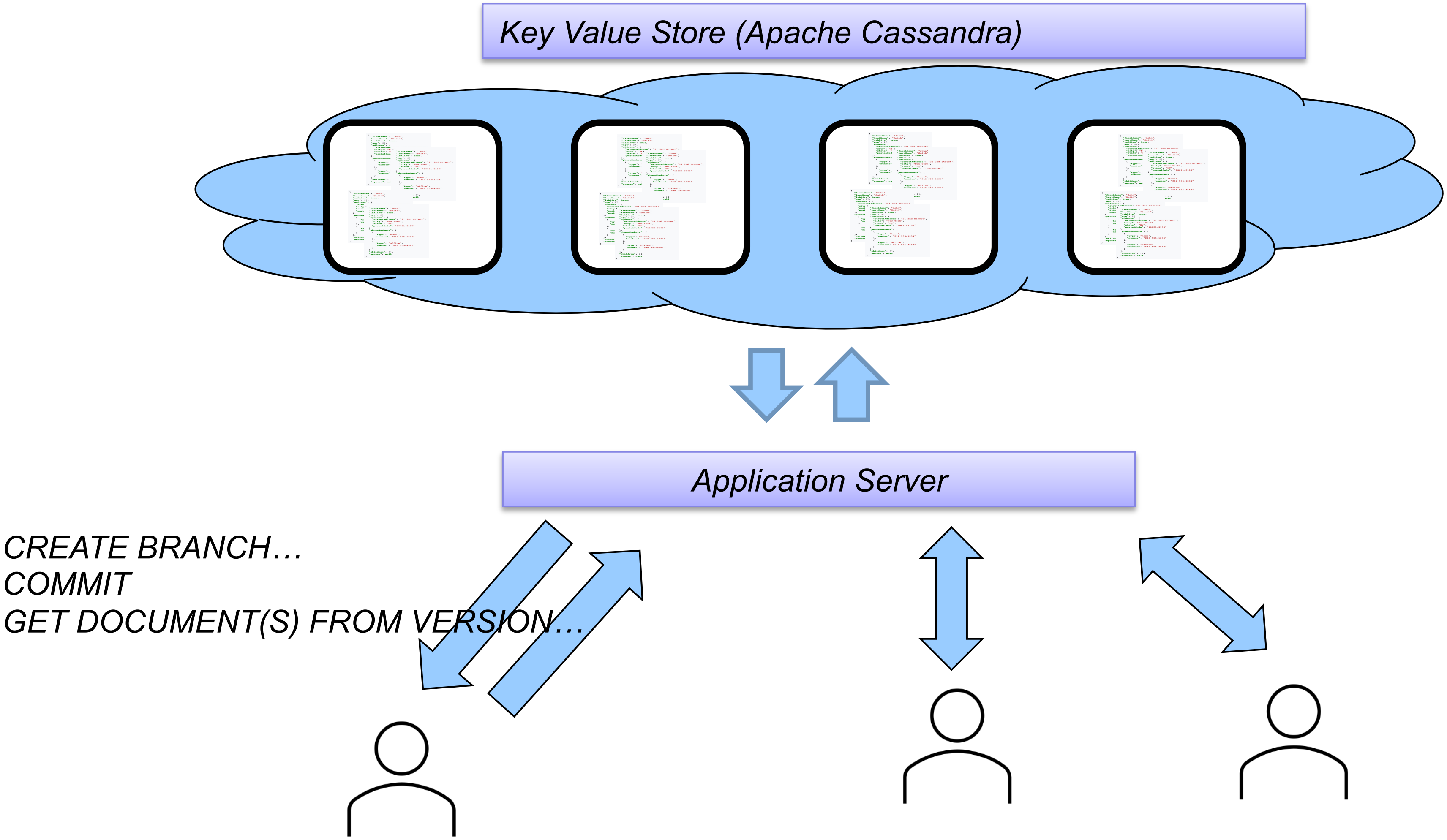
Minimum Cost Arborescence (MCA)
Edmonds’ algorithm
Time complexity = O(E + V logV)

Shortest Path Tree (SPT)
Dijkstra’s algorithm
Time complexity = O(E logV)

Scenario 3: Distributed Document Store

RStore

Designed as a wrapper on top of a key-value store to support versioning
Key design goal of not modifying the key-value store



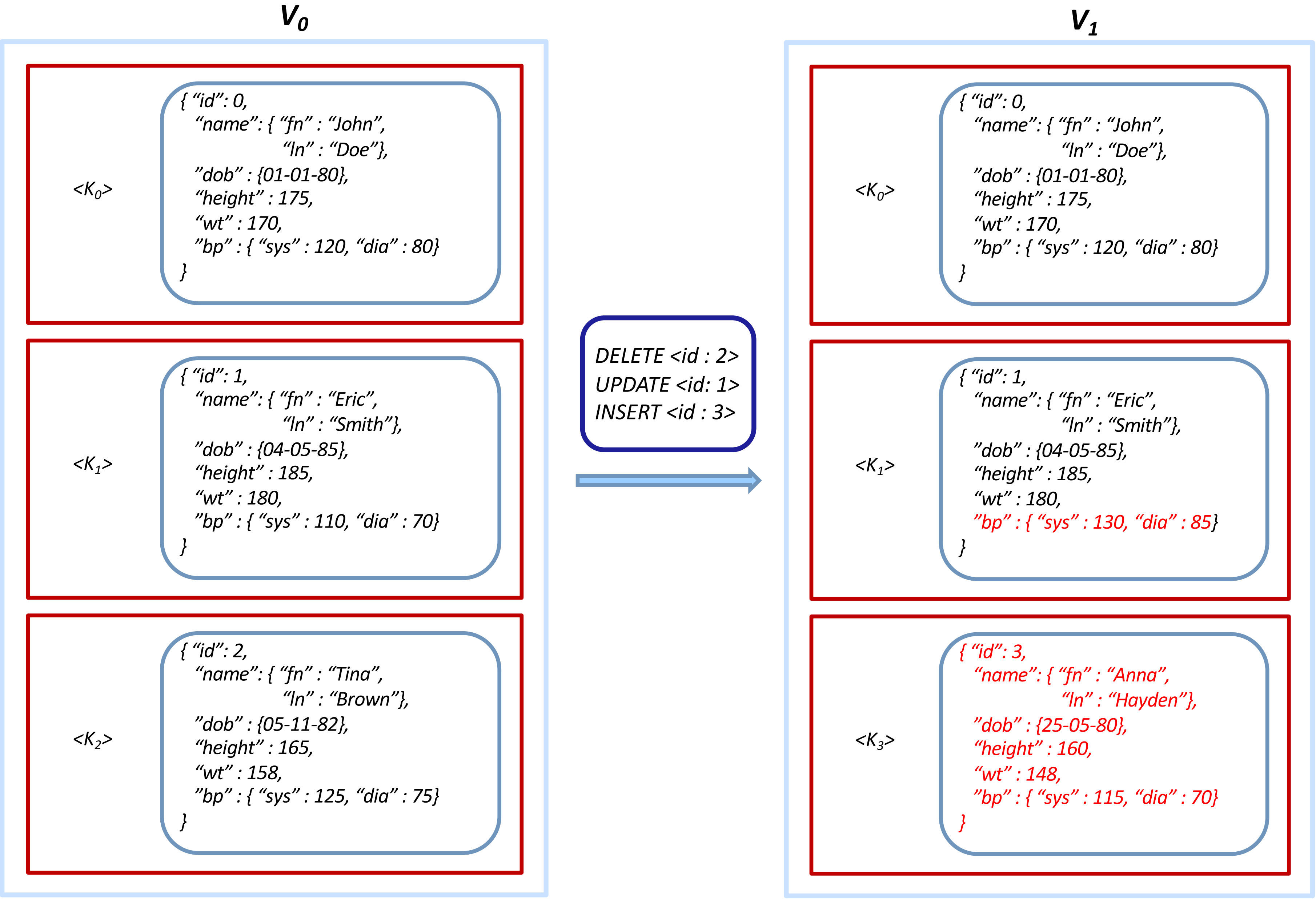
Key Value Store (Apache Cassandra)

Application Server

CREATE BRANCH…
COMMIT
GET DOCUMENT(S) FROM VERSION…

[SoCC'17, ICDE'18]

Data Model



V_0

$<K_0>$

```
{ “id”: 0,
  “name”: { “fn” : “John”,
            “ln” : “Doe”},
  ”dob” : {01-01-80},
  “height” : 175,
  “wt” : 170,
  ”bp” : { “sys” : 120, “dia” : 80}
}
```

$<K_1>$

```
{ “id”: 1,
  “name”: { “fn” : “Eric”,
            “ln” : “Smith”},
  ”dob” : {04-05-85},
  “height” : 185,
  “wt” : 180,
  ”bp” : { “sys” : 110, “dia” : 70}
}
```

$<K_2>$

```
{ “id”: 2,
  “name”: { “fn” : “Tina”,
            “ln” : “Brown”},
  ”dob” : {05-11-82},
  “height” : 165,
  “wt” : 158,
  ”bp” : { “sys” : 125, “dia” : 75}
}
```

DELETE <id : 2>
UPDATE <id: 1>
INSERT <id : 3>

V_1

$<K_0>$

```
{ “id”: 0,
  “name”: { “fn” : “John”,
            “ln” : “Doe”},
  ”dob” : {01-01-80},
  “height” : 175,
  “wt” : 170,
  ”bp” : { “sys” : 120, “dia” : 80}
}
```

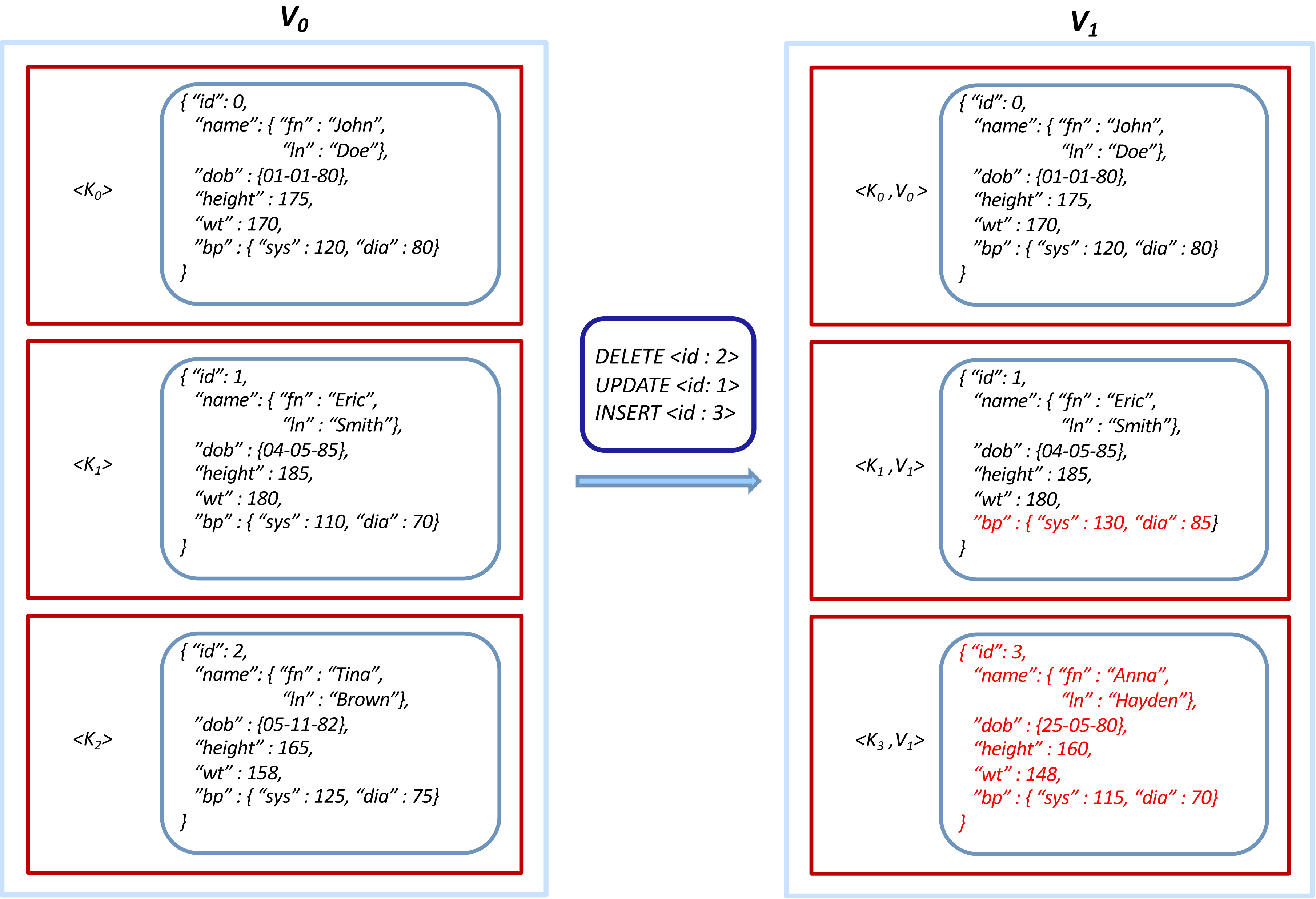
$<K_1>$

```
{ “id”: 1,
  “name”: { “fn” : “Eric”,
            “ln” : “Smith”},
  ”dob” : {04-05-85},
  “height” : 185,
  “wt” : 180,
  ”bp” : { “sys” : 130, “dia” : 85}
}
```

$<K_3>$

```
{ “id”: 3,
  “name”: { “fn” : “Anna”,
            “ln” : “Hayden”},
  ”dob” : {25-05-80},
  “height” : 160,
  “wt” : 148,
  ”bp” : { “sys” : 115, “dia” : 70}
}
```

Data Model: Composite Keys



V_0

<K_0>

```
{ "id": 0,
  "name": { "fn" : "John",
            "ln" : "Doe"},
  "dob" : {01-01-80},
  "height" : 175,
  "wt" : 170,
  "bp" : { "sys" : 120, "dia" : 80}
}
```

<K_1>

```
{ "id": 1,
  "name": { "fn" : "Eric",
            "ln" : "Smith"},
  "dob" : {04-05-85},
  "height" : 185,
  "wt" : 180,
  "bp" : { "sys" : 110, "dia" : 70}
}
```

<K_2>

```
{ "id": 2,
  "name": { "fn" : "Tina",
            "ln" : "Brown"},
  "dob" : {05-11-82},
  "height" : 165,
  "wt" : 158,
  "bp" : { "sys" : 125, "dia" : 75}
}
```

DELETE <id : 2>
UPDATE <id: 1>
INSERT <id : 3>

V_1

<K_0 ,V_0 >

```
{ "id": 0,
  "name": { "fn" : "John",
            "ln" : "Doe"},
  "dob" : {01-01-80},
  "height" : 175,
  "wt" : 170,
  "bp" : { "sys" : 120, "dia" : 80}
}
```

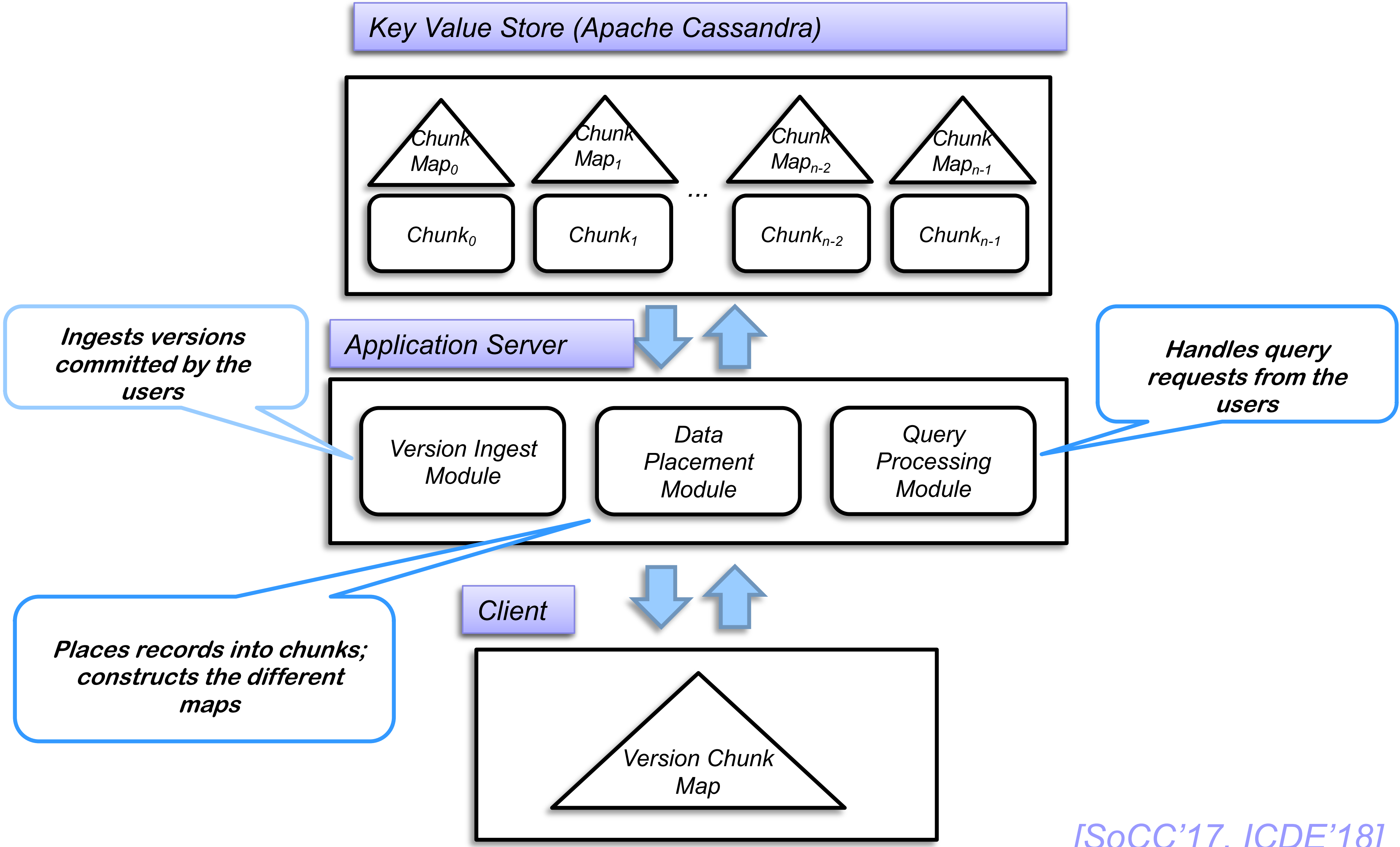
<K_1 ,V_1>

```
{ "id": 1,
  "name": { "fn" : "Eric",
            "ln" : "Smith"},
  "dob" : {04-05-85},
  "height" : 185,
  "wt" : 180,
  "bp" : { "sys" : 130, "dia" : 85}
}
```

<K_3 ,V_1>

```
{ "id": 3,
  "name": { "fn" : "Anna",
            "ln" : "Hayden"},
  "dob" : {25-05-80},
  "height" : 160,
  "wt" : 148,
  "bp" : { "sys" : 115, "dia" : 70}
}
```

RStore: Architecture



Ingests versions committed by the users

Handles query requests from the users

Places records into chunks; constructs the different maps

[SoCC'17, ICDE'18]

RStore: Overview

- Designed to support a wide range of retrieval queries, including partial version retrieval

- Based on creating chunks of similar records to minimize storage footprint
  - Employs several different partitioning algorithms to create chunks

- Results in much fewer queries to the back-end key value store
  - … by minimizing the number of chunks that a version spans

[SoCC'17, ICDE'18]

Outline

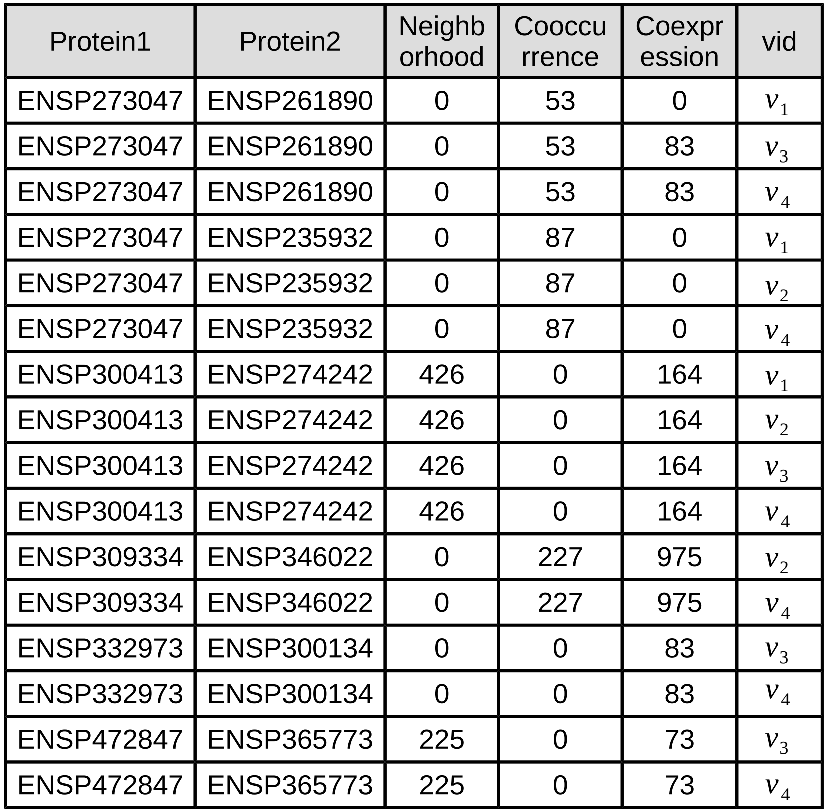
- DataHub: Overview
- OrpheusDB
- TardisDB
- Forkbase

Motivation

- Database systems don’t support versioning → entire datasets get copied during collaborative work
  - e.g., gene annotation datasets, or protein interaction networks
- OrpheusDB: Bolt-on versioning for RDBMS
  - Support versioning on top of an RDBMS, without modifications
  - Allow standard SQL-based querying of the tables within the versions

Storage Options (1)

a. Table with Versioned Records



Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vid
ENSP273047	ENSP261890	0	53	0	v_1
ENSP273047	ENSP261890	0	53	83	v_3
ENSP273047	ENSP261890	0	53	83	v_4
ENSP273047	ENSP235932	0	87	0	v_1
ENSP273047	ENSP235932	0	87	0	v_2
ENSP273047	ENSP235932	0	87	0	v_4
ENSP300413	ENSP274242	426	0	164	v_1
ENSP300413	ENSP274242	426	0	164	v_2
ENSP300413	ENSP274242	426	0	164	v_3
ENSP300413	ENSP274242	426	0	164	v_4
ENSP309334	ENSP346022	0	227	975	v_2
ENSP309334	ENSP346022	0	227	975	v_4
ENSP332973	ENSP300134	0	0	83	v_3
ENSP332973	ENSP300134	0	0	83	v_4
ENSP472847	ENSP365773	225	0	73	v_3
ENSP472847	ENSP365773	225	0	73	v_4

- Simple and supports querying individual versions
- High duplication -- a tuple in 100 versions is copied 100 times
- A simple “branch” requires a full copy of the tuples in that version

- Approach taken by temporal databases
    - Store a timestamp with each tuple
    - Doesn’t work with branching etc.

Storage Options (2)

b. Combined Table

Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vlist
ENSP273047	ENSP261890	0	53	0	$\{v_1\}$
ENSP273047	ENSP261890	0	53	83	$\{v_3, v_4\}$
ENSP273047	ENSP235932	0	87	0	$\{v_1, v_2, v_4\}$
ENSP300413	ENSP274242	426	0	164	$\{v_1, v_2, v_3, v_4\}$
ENSP309334	ENSP346022	0	227	975	$\{v_2, v_4\}$
ENSP332973	ENSP300134	0	0	83	$\{v_3, v_4\}$
ENSP472847	ENSP365773	225	0	73	$\{v_3, v_4\}$

data attributes

versioning attribute

- Requires efficient support for querying over arrays
- A simple “branch” requires modifying the arrays for all tuples in that version

Storage Options (3)

- Separate out the versioning information in a different set of tables
- Need to do a join to retrieve the version information
- Option 1: store a version list each record
  - A new version will require updating many tuples
- Option 2: store a record list with each version

c. Data Table + Versioning Table

rid	Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression
r_1	ENSP273047	ENSP261890	0	53	0
r_2	ENSP273047	ENSP235932	0	87	0
r_3	ENSP300413	ENSP274242	426	0	164
r_4	ENSP309334	ENSP346022	0	227	975
r_5	ENSP273047	ENSP261890	0	53	83
r_6	ENSP332973	ENSP300134	0	0	83
r_7	ENSP472847	ENSP365773	225	0	73

$\bowtie_\theta$

rid	vlist
r_1	$\{v_1\}$
r_2	$\{v_1, v_2, v_4\}$
r_3	$\{v_1, v_2, v_3, v_4\}$
r_4	$\{v_2, v_4\}$
r_5	$\{v_3, v_4\}$
r_6	$\{v_3, v_4\}$
r_7	$\{v_3, v_4\}$

c.i. Split-by-vlist

vid	rlist
v_1	$\{r_1, r_2, r_3\}$
v_2	$\{r_2, r_3, r_4\}$
v_3	$\{r_3, r_5, r_6, r_7\}$
v_4	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$

c.ii. Split-by-rlist

OrpheusDB Version Control API

- Collaborative Versioned Dataset (CVD)
  - A relation + versions of that relation
  - Version graph: DAG that maintains derivation information
  - All tuples/records in a CVD are “immutable”
  - Each relation assumed to have a ”primary key”

- APIs:
  - checkout: materialize a version as a regular table within the database
    - Only the user who issue checkout has access to the table
    - Can support “merge” operation to generate a single table as a union of multiple versions of the table

OrpheusDB Version Control API

- APIs:
  - commit: Add a modified table as new version to the CVD
    - Need to figure out which records changed from the parent (original) version
      - Use “primary key” for this purpose
      - Any changes from the parent version result in a new records in the CVD (all records are immutable in the CVD)
    - If `checkout` was done with multiple versions, then the new version has all of those as parents
  - Can do checkout to, and commit from, a CSV file
    - Need additional information to do the mappings
  - diff: compare two version and output the difference
  - init, create_user, config, etc…

OrpheusDB Version Control API

- SQL Commands
  - Can directly run SQL queries on specific version, without having to materialize it

```
SELECT ... FROM VERSION [vid] OF CVD [cvd], ...
```

```
SELECT * FROM VERSION 1, 2 OF CVD Interaction
WHERE coexpression > 80 LIMIT 50;
```

- Additional constructs to apply an aggregate across versions, identify versions with a specific property, etc.

System Architecture

- Implemented as a layer on top of a relational database

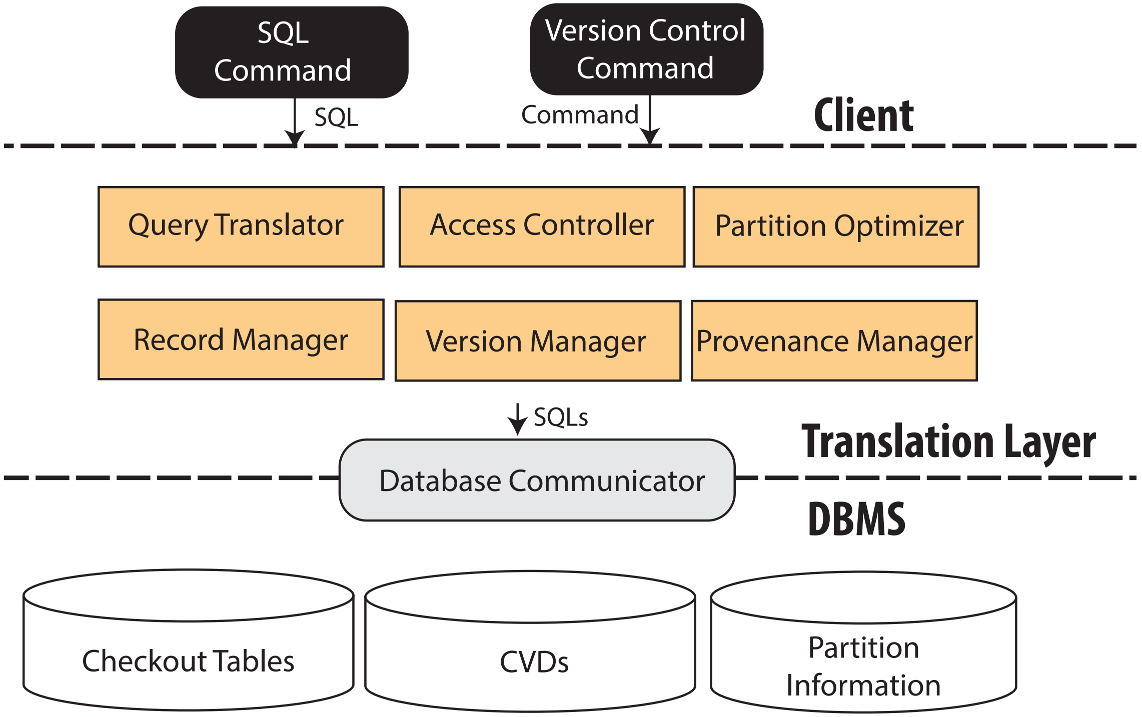
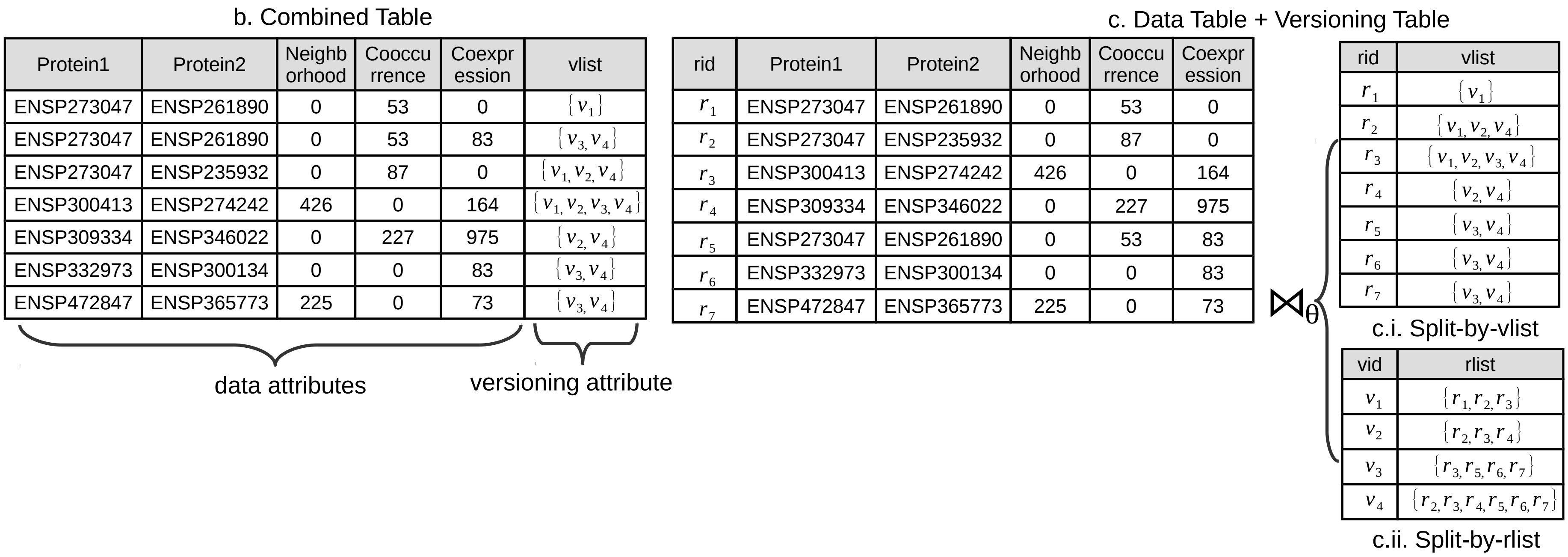


Figure 2: OrpheusDB Architecture

Storing CVDs

- Five approaches
  - Combined table (1(b))
  - Split-by-vlist
  - Split-by-rlist



b. Combined Table

Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression	vlist
ENSP273047	ENSP261890	0	53	0	$\{v_1\}$
ENSP273047	ENSP261890	0	53	83	$\{v_3, v_4\}$
ENSP273047	ENSP235932	0	87	0	$\{v_1, v_2, v_4\}$
ENSP300413	ENSP274242	426	0	164	$\{v_1, v_2, v_3, v_4\}$
ENSP309334	ENSP346022	0	227	975	$\{v_2, v_4\}$
ENSP332973	ENSP300134	0	0	83	$\{v_3, v_4\}$
ENSP472847	ENSP365773	225	0	73	$\{v_3, v_4\}$

data attributes

versioning attribute

c. Data Table + Versioning Table

rid	Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression
r_1	ENSP273047	ENSP261890	0	53	0
r_2	ENSP273047	ENSP235932	0	87	0
r_3	ENSP300413	ENSP274242	426	0	164
r_4	ENSP309334	ENSP346022	0	227	975
r_5	ENSP273047	ENSP261890	0	53	83
r_6	ENSP332973	ENSP300134	0	0	83
r_7	ENSP472847	ENSP365773	225	0	73

$\bowtie_\theta$

rid	vlist
r_1	$\{v_1\}$
r_2	$\{v_1, v_2, v_4\}$
r_3	$\{v_1, v_2, v_3, v_4\}$
r_4	$\{v_2, v_4\}$
r_5	$\{v_3, v_4\}$
r_6	$\{v_3, v_4\}$
r_7	$\{v_3, v_4\}$

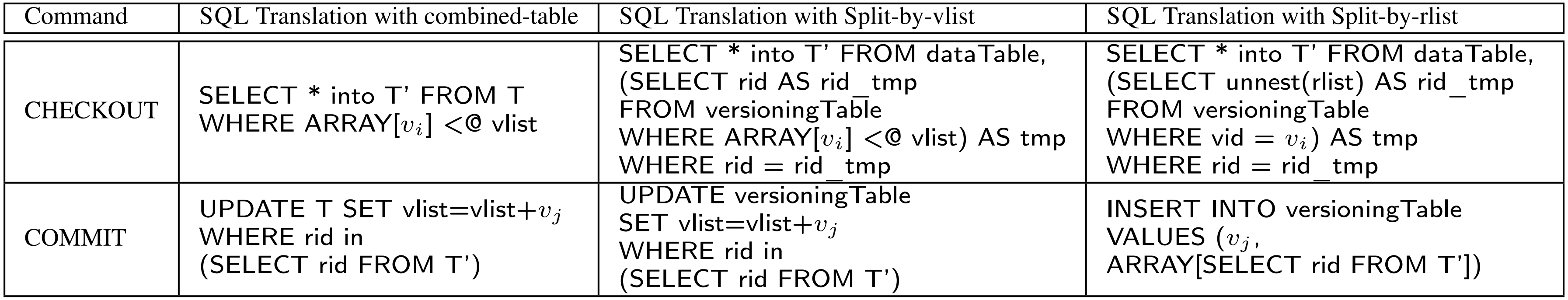
c.i. Split-by-vlist

vid	rlist
v_1	$\{r_1, r_2, r_3\}$
v_2	$\{r_2, r_3, r_4\}$
v_3	$\{r_3, r_5, r_6, r_7\}$
v_4	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$

c.ii. Split-by-rlist

Storing CVDs

- Five approaches
  - Combined table (1(b))
  - Split-by-vlist
  - Split-by-rlist



Command	SQL Translation with combined-table	SQL Translation with Split-by-vlist	SQL Translation with Split-by-rlist
CHECKOUT	SELECT * into T' FROM T WHERE ARRAY[v_i] <@ vlist	SELECT * into T' FROM dataTable, (SELECT rid AS rid_tmp FROM versioningTable WHERE ARRAY[v_i] <@ vlist) AS tmp WHERE rid = rid_tmp	SELECT * into T' FROM dataTable, (SELECT unnest(rlist) AS rid_tmp FROM versioningTable WHERE vid = v_i) AS tmp WHERE rid = rid_tmp
COMMIT	UPDATE T SET vlist=vlist+v_j WHERE rid in (SELECT rid FROM T')	UPDATE versioningTable SET vlist=vlist+v_j WHERE rid in (SELECT rid FROM T')	INSERT INTO versioningTable VALUES (v_j, ARRAY[SELECT rid FROM T'])

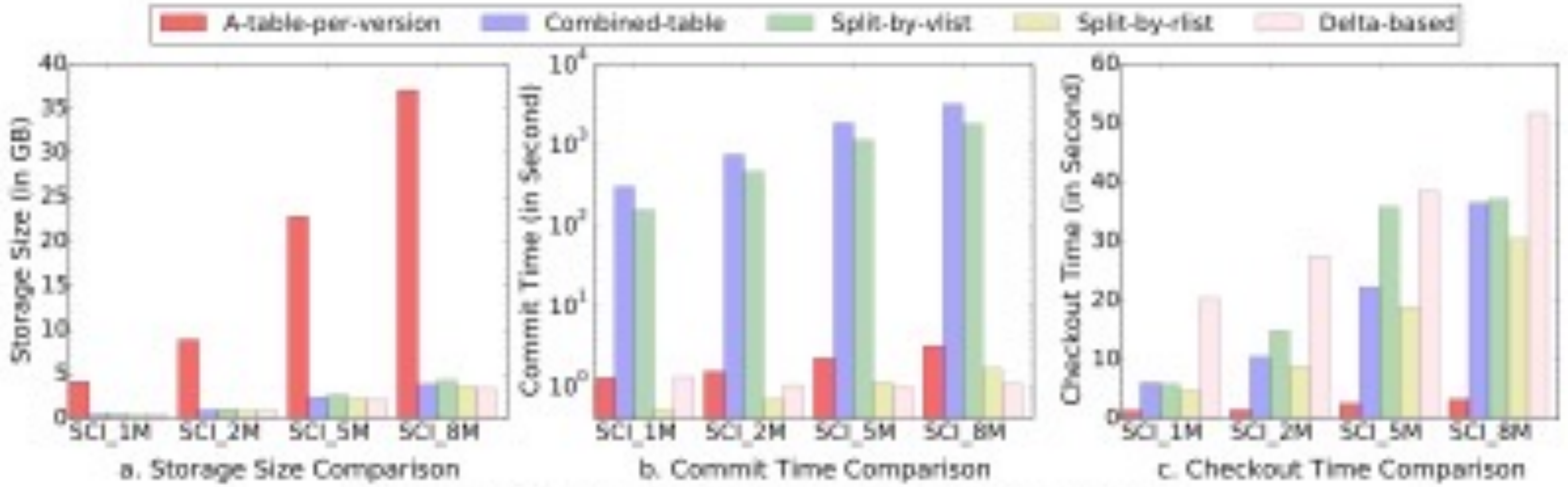
Table 1: SQL Queries for Checkout and Commit Commands with Different Data Models

Storing CVDs

- Five approaches
  - Combined table (1(b))
  - Split-by-vlist
  - Split-by-rlist
  - Delta-based approach (also called “version-first”)
    - Store each version as a “delta” from one of its parent versions
    - Need a new regular table for each version
    - Lower storage space if most changes are local
    - Harder to do queries
  - A-Table-Per-Version (naïve baseline)

Comparing the Options

- No single winner
- Split-by-rlist provides best balance



a. Storage Size Comparison

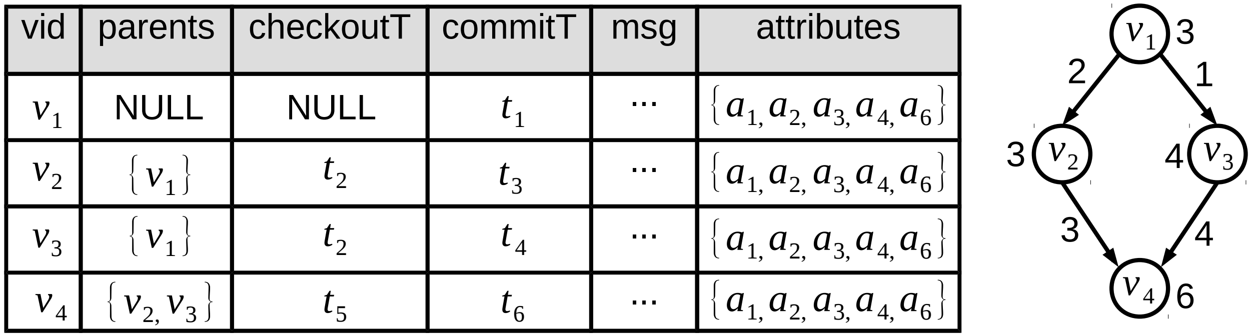
b. Commit Time Comparison

c. Checkout Time Comparison

Figure 3: Comparison Between Different Data Models

Version Derivation Metadata

- Version-level provenance maintained in a metadata table
- Supports “schema changes” during commit
  - Somewhat simplistic -- hard to handle this in general



vid	parents	checkoutT	commitT	msg	attributes
v_1	NULL	NULL	t_1	⋯	$\{a_1, a_2, a_3, a_4, a_6\}$
v_2	$\{v_1\}$	t_2	t_3	⋯	$\{a_1, a_2, a_3, a_4, a_6\}$
v_3	$\{v_1\}$	t_2	t_4	⋯	$\{a_1, a_2, a_3, a_4, a_6\}$
v_4	$\{v_2, v_3\}$	t_5	t_6	⋯	$\{a_1, a_2, a_3, a_4, a_6\}$

a. Metadata Table

b. Version Graph

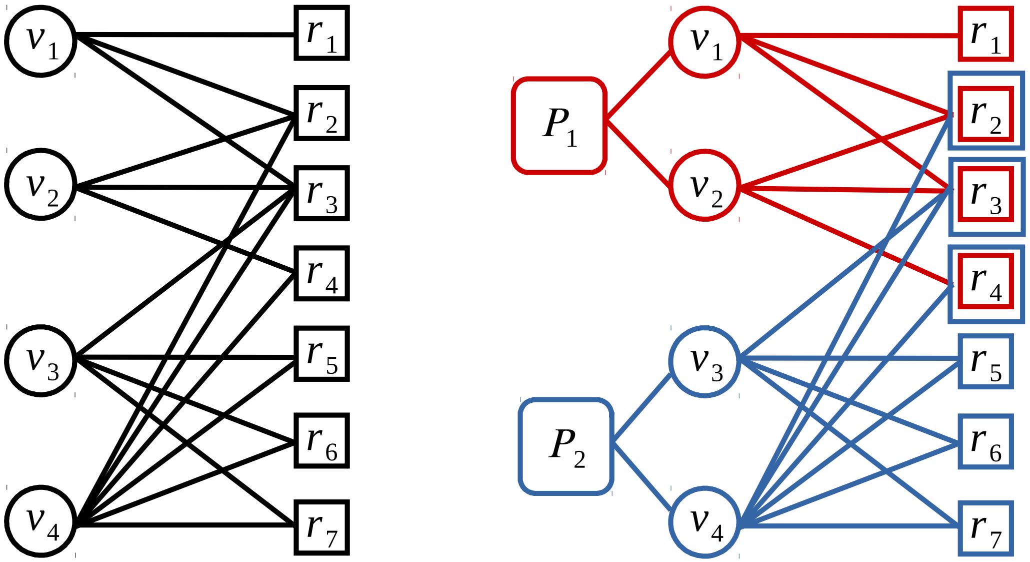
Figure 4: Metadata Table and Version Graph (Fixed Schema)

Optimization Problem

- Too much redundant processing when checking out a version if..
  - .. number of records in the version << total number of records
- Use ”Partitioning”
  - e.g., imagine 100 versions
    - 10 versions, each containing a large fraction of t1, …, t_100
    - 10 versions, each containing a large fraction of t_101, …, t_200
    - …
  - If all stored together, then checking out a version requires processing 100 * 100 = 10000 records
  - If stored in groups of 10 versions, then checking out requires processing only 100 records
- In general, won’t find such ”clean” partitioning
  - But, depending on the datasets, it might still provide significant benefits
- Also partitioning increases total storage cost

Optimization Problem

- Problem is too hard to solve optimally
- Instead, design efficient heuristics



a. Bipartite Graph b. Illustration of Partitioning

Figure 6: Version-Record Bipartite Graph & Partitioning

Outline

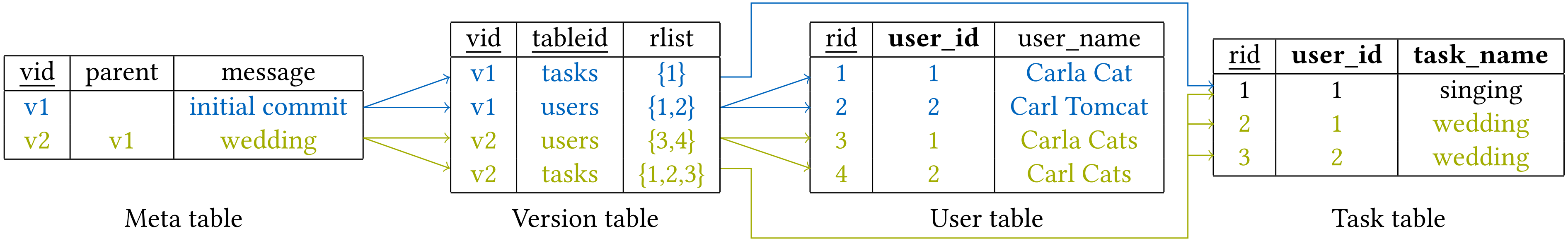
- DataHub: Overview
- OrpheusDB
- TardisDB
- Forkbase

Overview

- Motivation analogous to OrpheusDB
  - Versioning within a relational database system
  - Supports many use cases that need to be done outside DBMS
- But:
  - Support multiple tables instead a single table per version
  - For a main-memory database system
- Paper also develops a benchmark for versioning based on Wikipedia

MusaeusDB

- Expands upon OrpheusDB data model, with keeping version information in a separate table
- Main difference:
  - Extra attribute “tableid” in the “version table” to allow for multiple tables



vid	parent	message
v1		initial commit
v2	v1	wedding

Meta table

vid	tableid	rlist
v1	tasks	{1}
v1	users	{1,2}
v2	users	{3,4}
v2	tasks	{1,2,3}

Version table

rid	**user_id**	user_name
1	1	Carla Cat
2	2	Carl Tomcat
3	1	Carla Cats
4	2	Carl Cats

User table

rid	**user_id**	**task_name**
1	1	singing
2	1	wedding
3	2	wedding

Task table

Figure 2: Schema: Version table and meta table for managing the commits on the left; tables containing the data on the right; the record id serves as a key for every tuple.

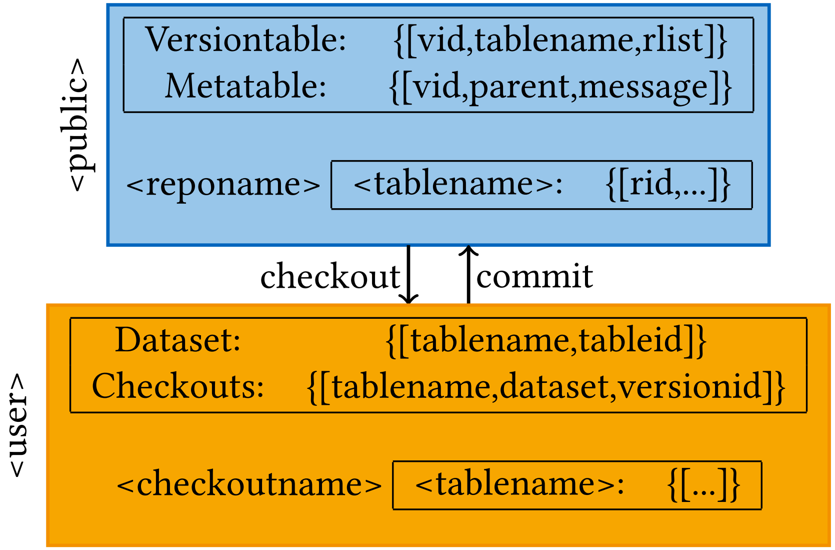
MusaeusDB

- Private namespaces for users when they checkout

init: add the requisite tables and attributes to an existing database for versioning

checkout: copies the tables to a private namespace

commit: update the global repository with changed/inserted/deleted tuples



<public>

Versiontable: {[vid,tablename,rlist]}
Metatable: {[vid,parent,message]}

<reponame> <tablename>: {[rid,...]}

checkout
commit

<user>

Dataset: {[tablename,tableid]}
Checkouts: {[tablename,dataset,versionid]}

<checkoutname> <tablename>: {[...]}

Figure 3: Distinction between global and local (user) space in *MusaeusDB*: The global space maintains a separate namespace for each repository, relations can be checked out for modifications in the user's namespace.

MusaeusSQL

- Unified interface on top

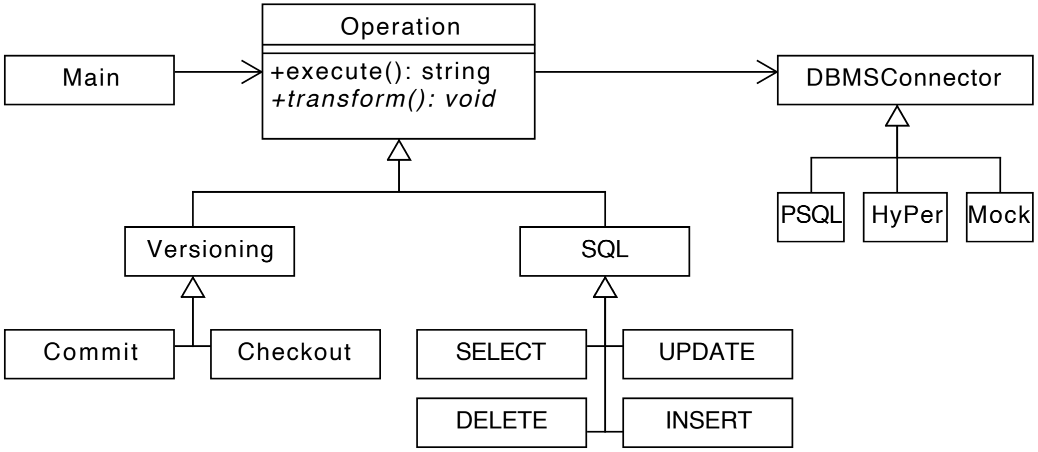
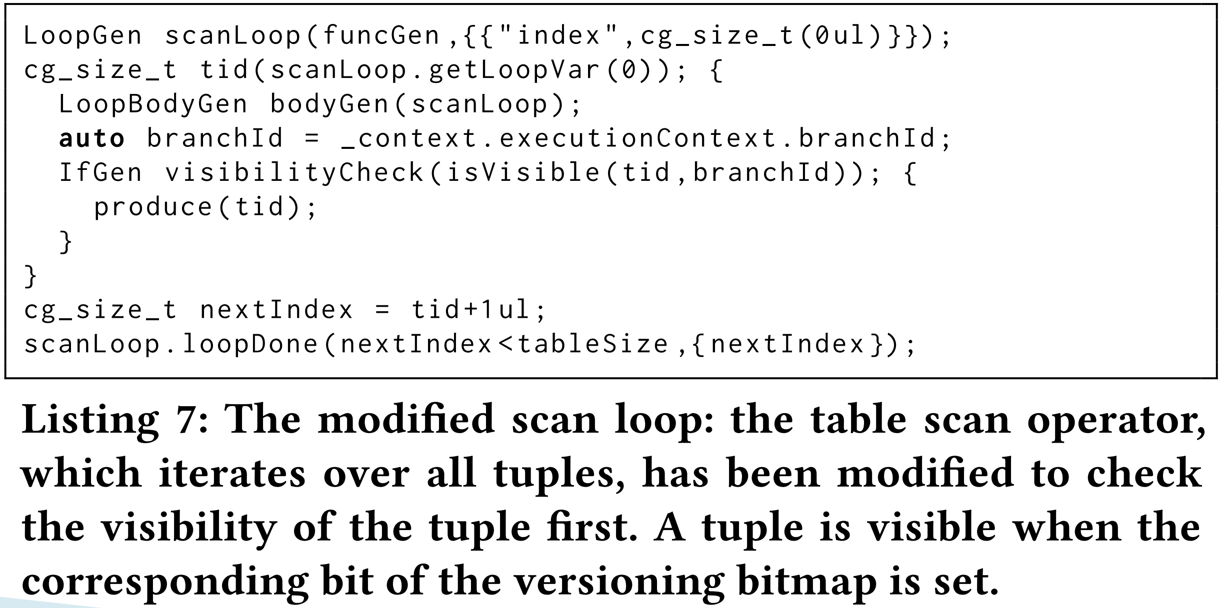


Figure 4: Architecture of *MusaeusSQL*: Operations are divided into basic SQL and versioning commands; SQL commands are transformed as the extended schema is hidden, versioning commands are translated into SQL queries.

TardisDB

- Integrated versioning into a main-memory system
- Uses the “tuple-first” approach from Decibel
  - Each tuple is associated with a bitmap telling which versions it belongs to
- For query processing, only the Scan operator changes

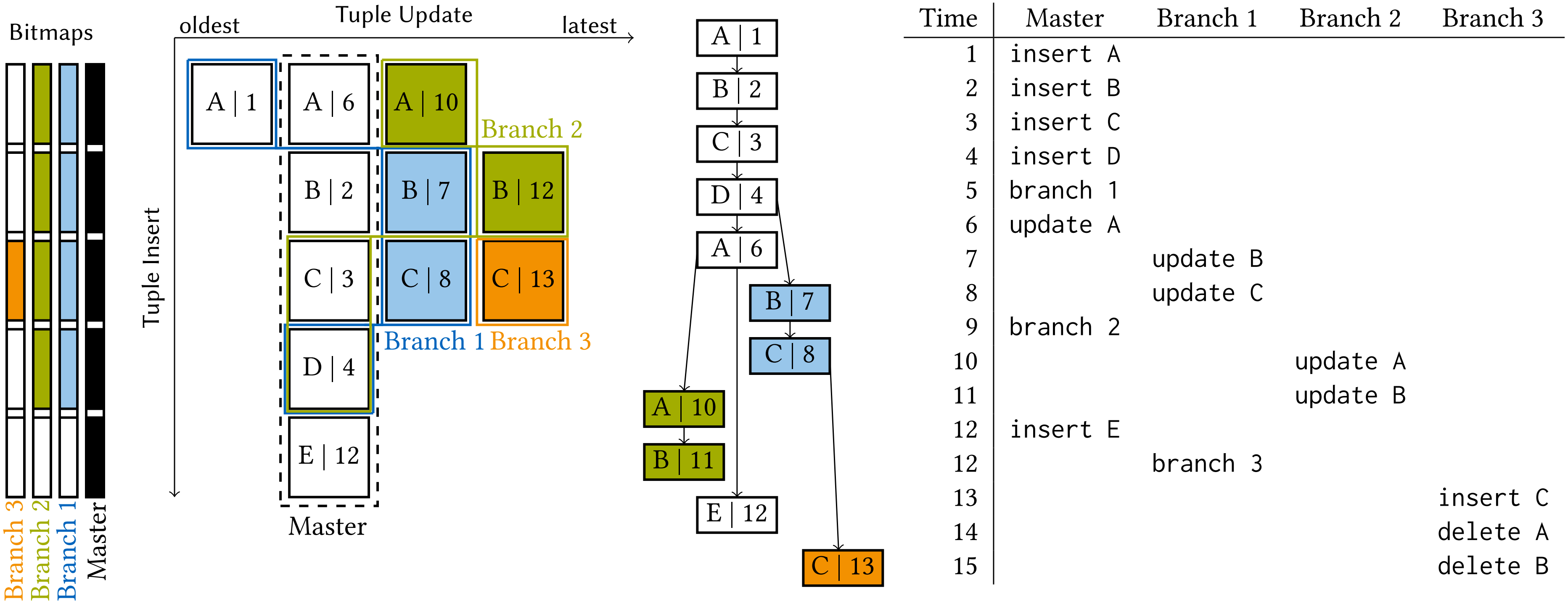


```
LoopGen scanLoop(funcGen,{{"index",cg_size_t(0ul)}});
cg_size_t tid(scanLoop.getLoopVar(0)); {
  LoopBodyGen bodyGen(scanLoop);
  auto branchId = _context.executionContext.branchId;
  IfGen visibilityCheck(isVisible(tid,branchId)); {
    produce(tid);
  }
}
cg_size_t nextIndex = tid+1ul;
scanLoop.loopDone(nextIndex<tableSize,{nextIndex});
```

Listing 7: The modified scan loop: the table scan operator, which iterates over all tuples, has been modified to check the visibility of the tuple first. A tuple is visible when the corresponding bit of the versioning bitmap is set.

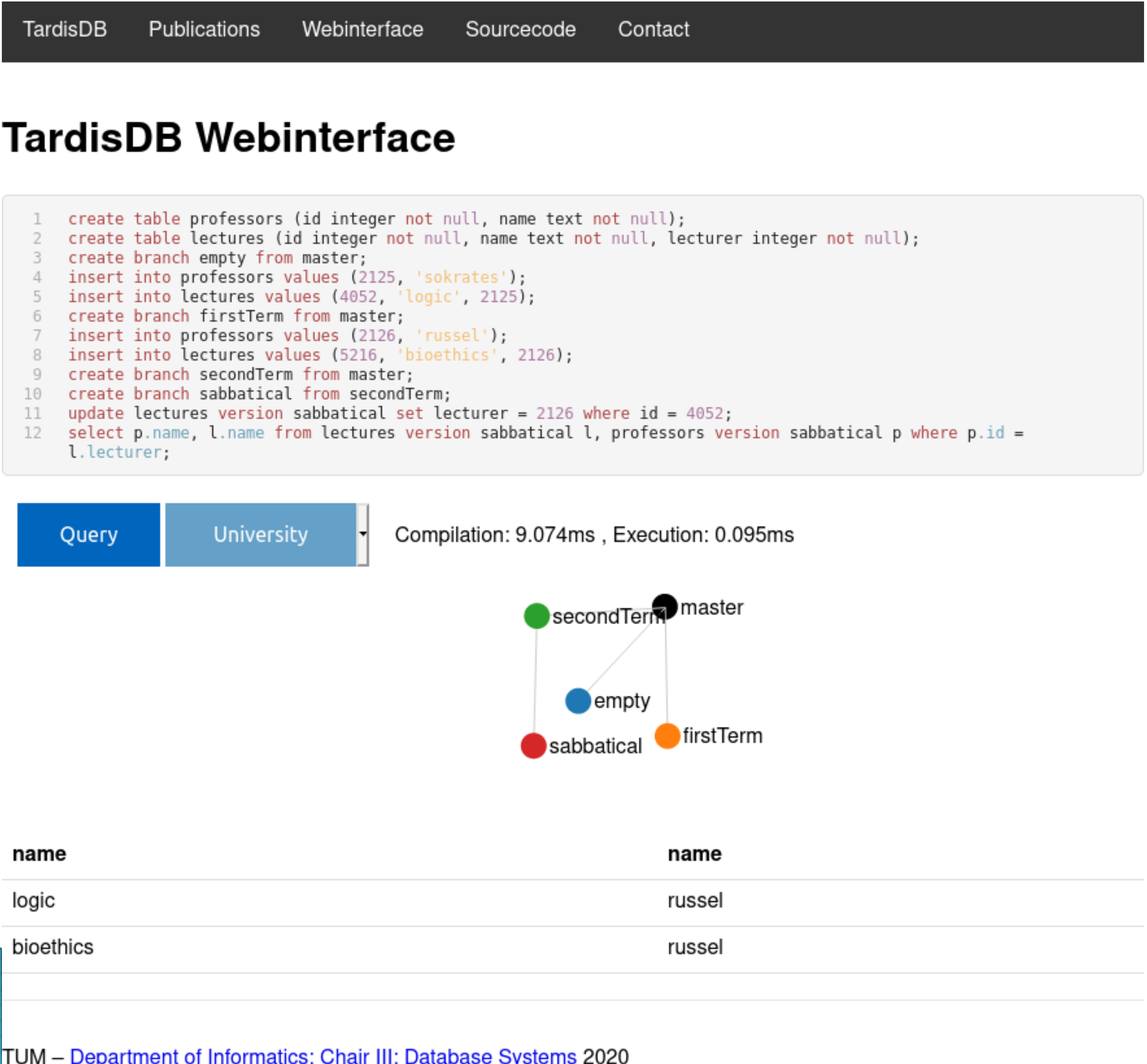
TardisDB

- Uses MVCC for the versioning



Time	Master	Branch 1	Branch 2	Branch 3
1	`insert A`			
2	`insert B`			
3	`insert C`			
4	`insert D`			
5	`branch 1`			
6	`update A`			
7		`update B`		
8		`update C`		
9	`branch 2`			
10			`update A`	
11			`update B`	
12	`insert E`			
12		`branch 3`		
13				`insert C`
14				`delete A`
15				`delete B`

Figure 6: Adaption of multi-version concurrency control for versioning (left): bitmaps for each branch indicate the included tuples; an insert increases the size of all bitmaps. Updates in the master branch are handled in place with a pointer to the previous version, updates from other branches are prepended. Tuples receive a unique timestamp, their colour indicates the creator branch. Descendance tree (middle) determines the tuple visibility for the corresponding history (right).



TardisDB Webinterface

```
create table professors (id integer not null, name text not null);
create table lectures (id integer not null, name text not null, lecturer integer not null);
create branch empty from master;
insert into professors values (2125, 'sokrates');
insert into lectures values (4052, 'logic', 2125);
create branch firstTerm from master;
insert into professors values (2126, 'russel');
insert into lectures values (5216, 'bioethics', 2126);
create branch secondTerm from master;
create branch sabbatical from secondTerm;
update lectures version sabbatical set lecturer = 2126 where id = 4052;
select p.name, l.name from lectures version sabbatical l, professors version sabbatical p where p.id = l.lecturer;
```

Compilation: 9.074ms , Execution: 0.095ms

name	name
logic	russel
bioethics	russel

TUM – Department of Informatics: Chair III: Database Systems 2020

Outline

- DataHub: Overview
- OrpheusDB
- TardisDB
- Forkbase

Motivation

- Many applications need a storage layer that support versioning and tamper-resistance
  - Collaborative applications (i.e., motivation for DataHub)
  - Blockchain systems (distributed tamperproof ledgers)

- Forkbase: a storage engine that:
  - Supports versioning and tamper-resistance
  - Splits up large objects into *data chunks* for deduplication
  - Support general “fork semantics” (branch and merge)
  - Simple APIs
  - Scales well to many nodes through two-layer partitioning

Forkbase Design

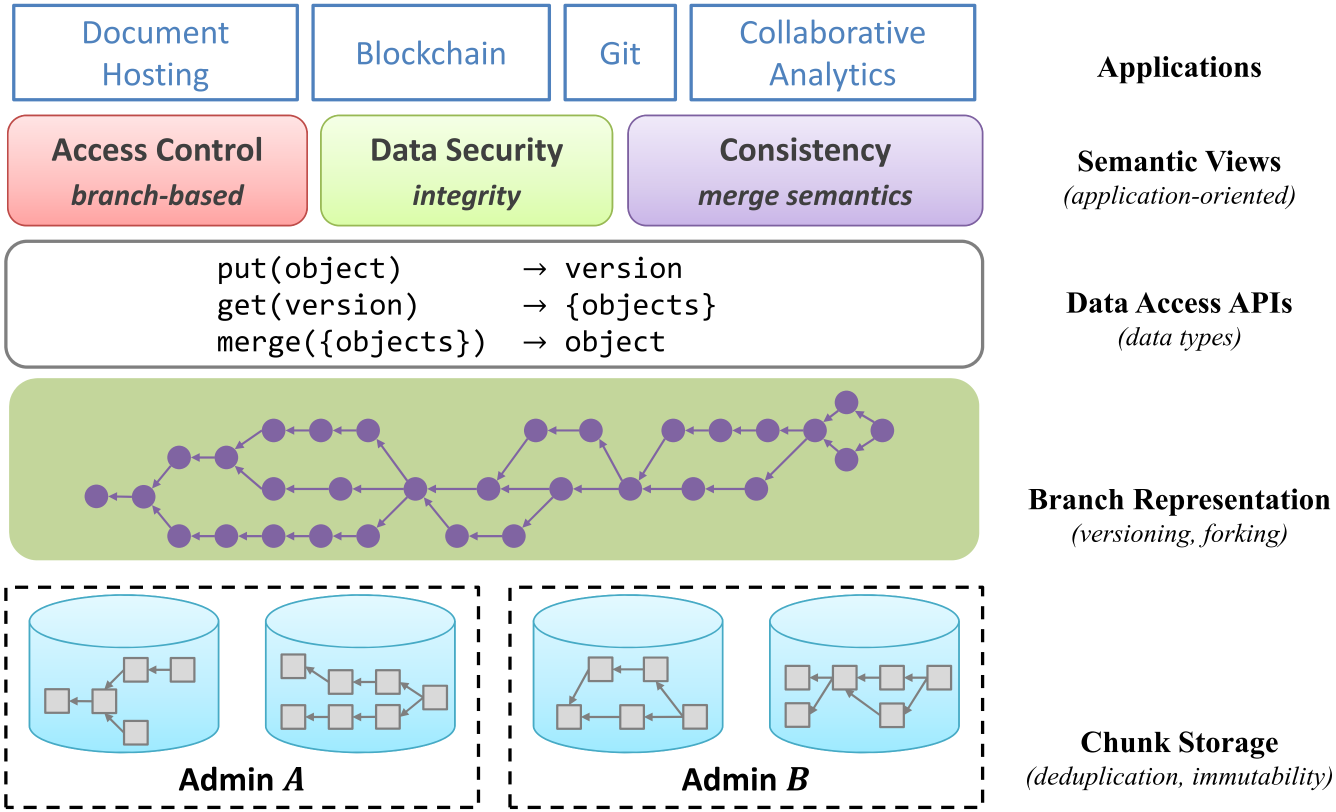
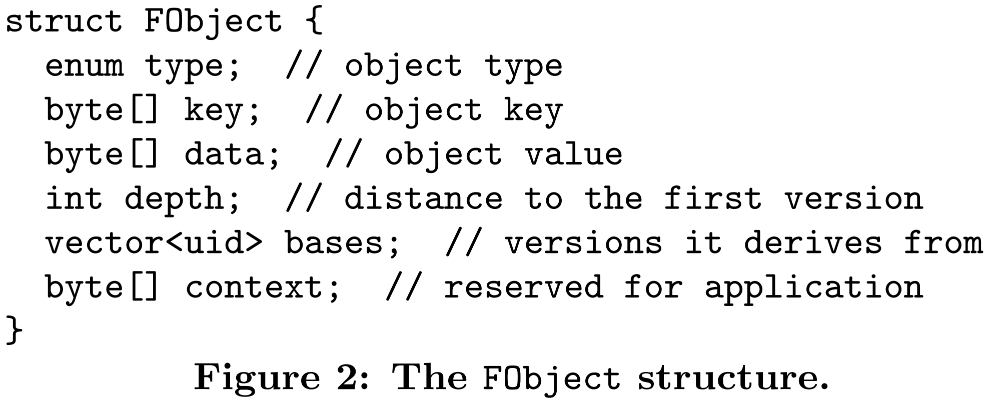


Figure 1: The *ForkBase* stack offers advanced features to various classes of modern applications.

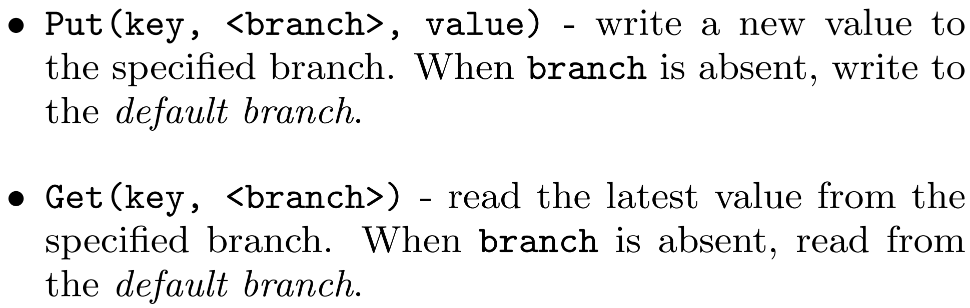
Data Model and APIs

- FObject: a generic object type that is versioned



```
struct FObject {
  enum type;  // object type
  byte[] key;  // object key
  byte[] data;  // object value
  int depth;  // distance to the first version
  vector<uid> bases;  // versions it derives from
  byte[] context;  // reserved for application
}
```

Figure 2: The `FObject` structure.

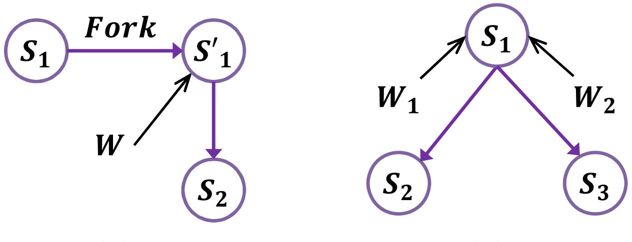


- `Put(key, <branch>, value)` - write a new value to the specified branch. When `branch` is absent, write to the *default branch*.
- `Get(key, <branch>)` - read the latest value from the specified branch. When `branch` is absent, read from the *default branch*.

Tamper resistance through linking versioning using a cryptographic hash chain (i.e., a blockchain)

Fork and Merge Operations

- FObject: a generic object type that is versioned



(a) (b)

Figure 3: Generic fork semantics supported for both (a) fork on demand and (b) fork on conflict.

```
ForkBaseConnector db;
// Put a blob to the default master branch
Blob blob {"my value"};
db.Put("my key", blob);
// Fork to a new branch
db.Fork("my key", "master", "new branch");

// Get the blob
FObject value = db.Get("my key", "new branch");
if (value.type() != Blob)
  throw TypeNotMatchError;
blob = value.Blob();

// Remove 10 bytes from beginning and append new
// Changes are buffered in client
blob.Remove(0, 10);
blob.Append("some more");
// Commit changes to that branch
db.Put("my key", "new branch", blob);
```

Implementation

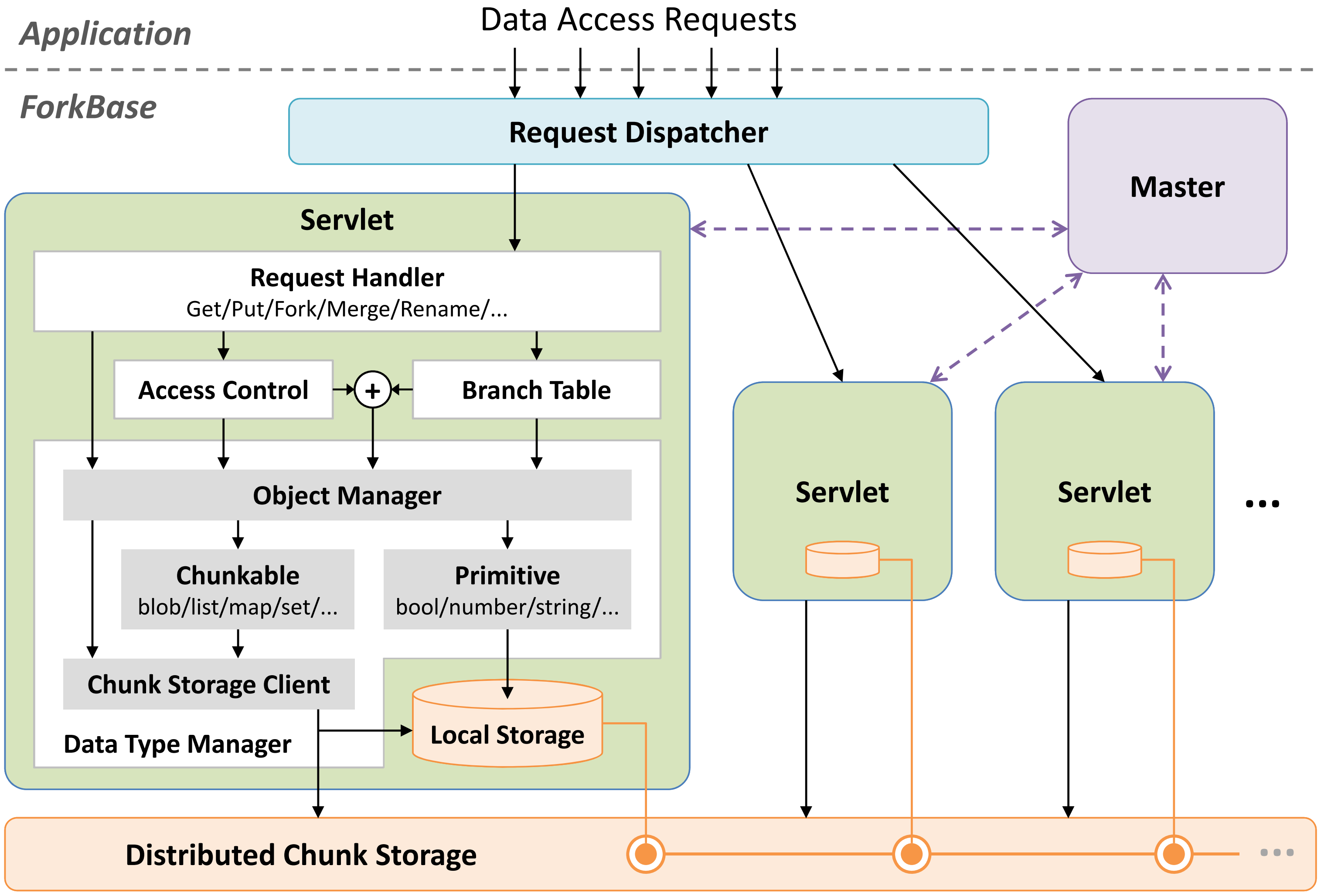


Figure 5: Architecture of a *ForkBase* cluster.

Pattern-Oriented-Splitting Tree

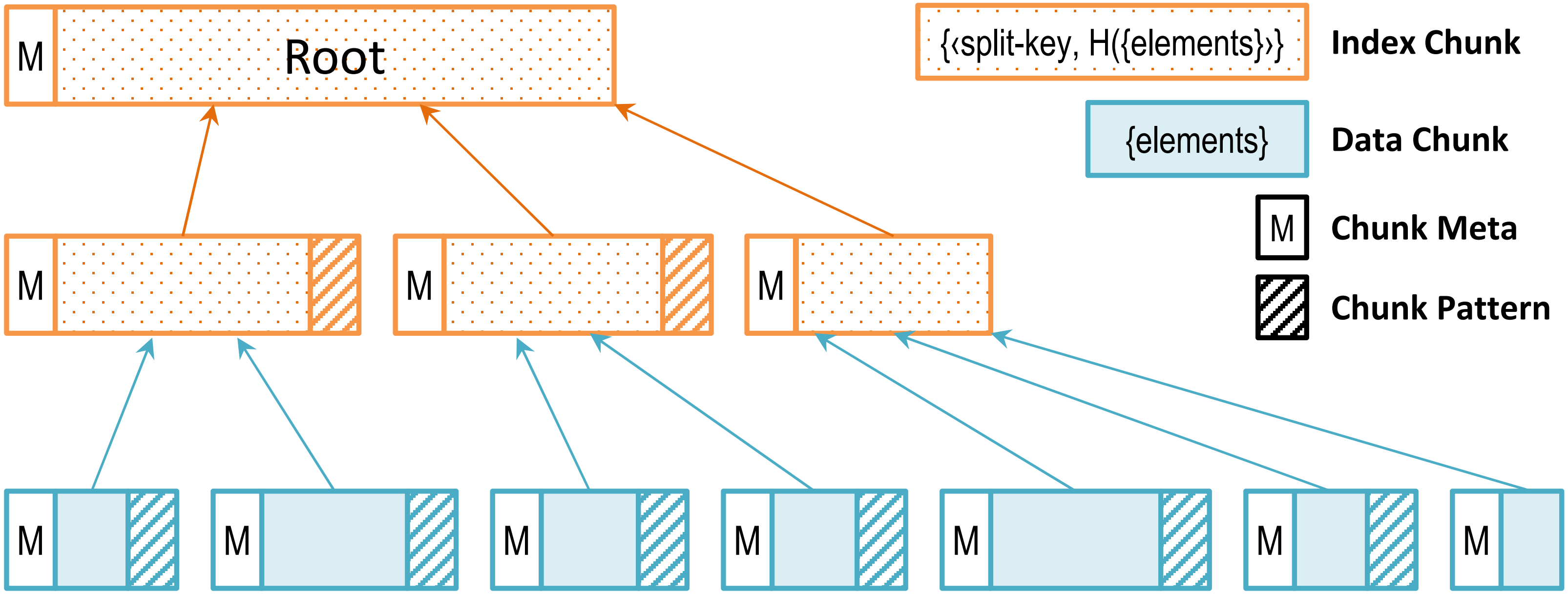


Figure 6: Pattern-Oriented-Splitting Tree (POS-tree) resembling a B^+-tree and Merkle tree.

Pattern-Oriented-Splitting Tree

- Leaf nodes are created through “content-based slicing”
  - Treat the data as sequence of bytes
  - Look for the first k-byte sequence that hashes to a fixed pattern (e.g., “…0000000”)
  - Create first leaf node that ends at that sequence
  - Look for the next k-byte sequence…
  - Use “rolling hashes” to speed this up (lot of work in storage deduplication)

- Index nodes use the same idea, but using the “cid” of the leaves instead of hashing
  - Those have some randomness properties since they are cryptographic hashes

Forkbase Use Cases

- Hyperledger Blockchain
  - Can replace the underlying state storage (Merkle Tree) with Forkbase

- Wiki Engine
  - For collaborative editing workflows
  - Can directly store the data into Forkbase

- Collaborative Analytics

Summary

- Immutability increasingly seen as a must-have in many data management systems
  - Versioning, tamper-resistance, fork/branch semantics etc.

- Many open challenges
  - Storage management, support for queries/transactions, schema evolution, analytics, …