CS4221
Cloud Databases III. OLAP optimizations

Yao LU

2024 Semester 2

National University of Singapore
School of Computing

Today’s agenda

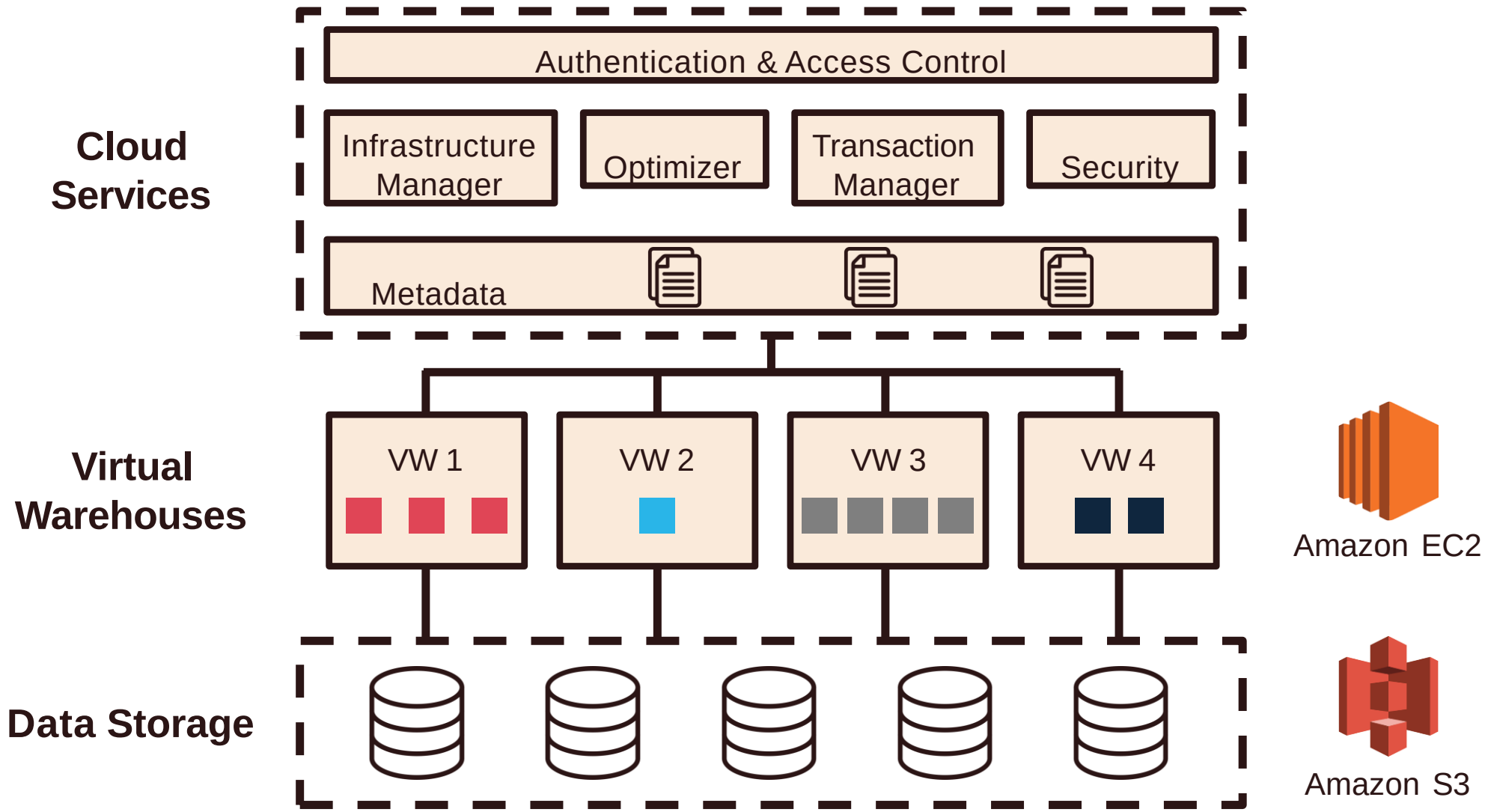
- Current practice in distributed OLAP
- Cost-intelligent data analytics in the cloud

The trend: disaggregated OLAP

- One recent trend of the last decade is the breakout of OLAP engine sub-systems into standalone open-source components.
  - This is typically done by organizations not in the business of selling DBMS software.

- **Examples:**
  - System Catalogs
  - Query Optimizers
  - File Format / Access Libraries
  - Execution Engines

Recall: Snowflake architecture



System catalogs

- A DBMS tracks a database's schema (table, columns) and data files in its catalog.
  - If the DBMS is on the data ingestion path, it can maintain the catalog incrementally.
  - If an external process adds data files, it also needs to update the catalog so that the DBMS is aware of them.

- Notable implementations:
  - HCatalog
  - Google Data Catalog
  - Amazon Glue Data Catalog

Execution engines

- Standalone libraries for executing vectorized query operators on columnar data.
  - Input is a DAG of physical operators.
  - Require external scheduling and orchestration.

- Notable implementations:
  - Velox
  - DataFusion
  - Intel OAP

Query optimizers

- Extendible search engine framework for heuristic- and cost-based query optimization.
  - Applications provide transformation rules and cost estimates.
  - Framework returns either a logical or physical query plan.
- This is the hardest part to build in any DBMS.

- Notable implementations:
  - Greenplum Orca
  - Apache Calcite

Query optimization for distributed execution

- All the optimizations that we talked about before are still applicable in a distributed environment.
  - Predicate Pushdown
  - Projection Pushdown
  - Optimal Join Orderings

- Distributed query optimization is even harder because it must consider the physical location of data and network transfer costs.
  - Using broadcast join vs. repartition join?
  - Considering the impact of data partitioning.

Distributed query execution

- Executing an OLAP query in a distributed DBMS is roughly the same as on a single-node DBMS.
  - A query plan is represented as a tree of physical operators.

- For each operator, the DBMS considers where input is coming from and where to send output, like parallel execution in a single node.

Unique challenges

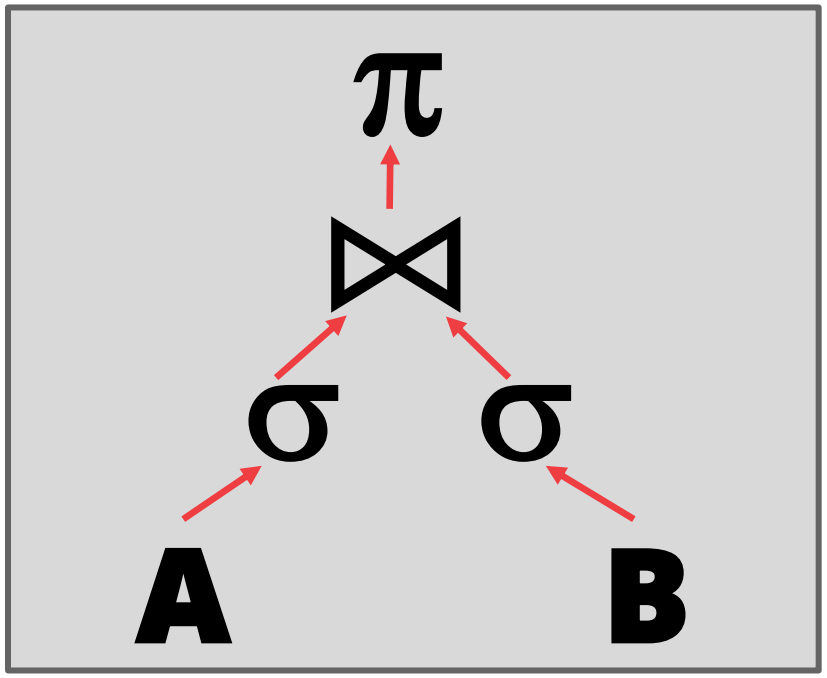
- Data is partitioned across nodes, so a worker thread does not have access to all data for free: accessing data requires network communication.

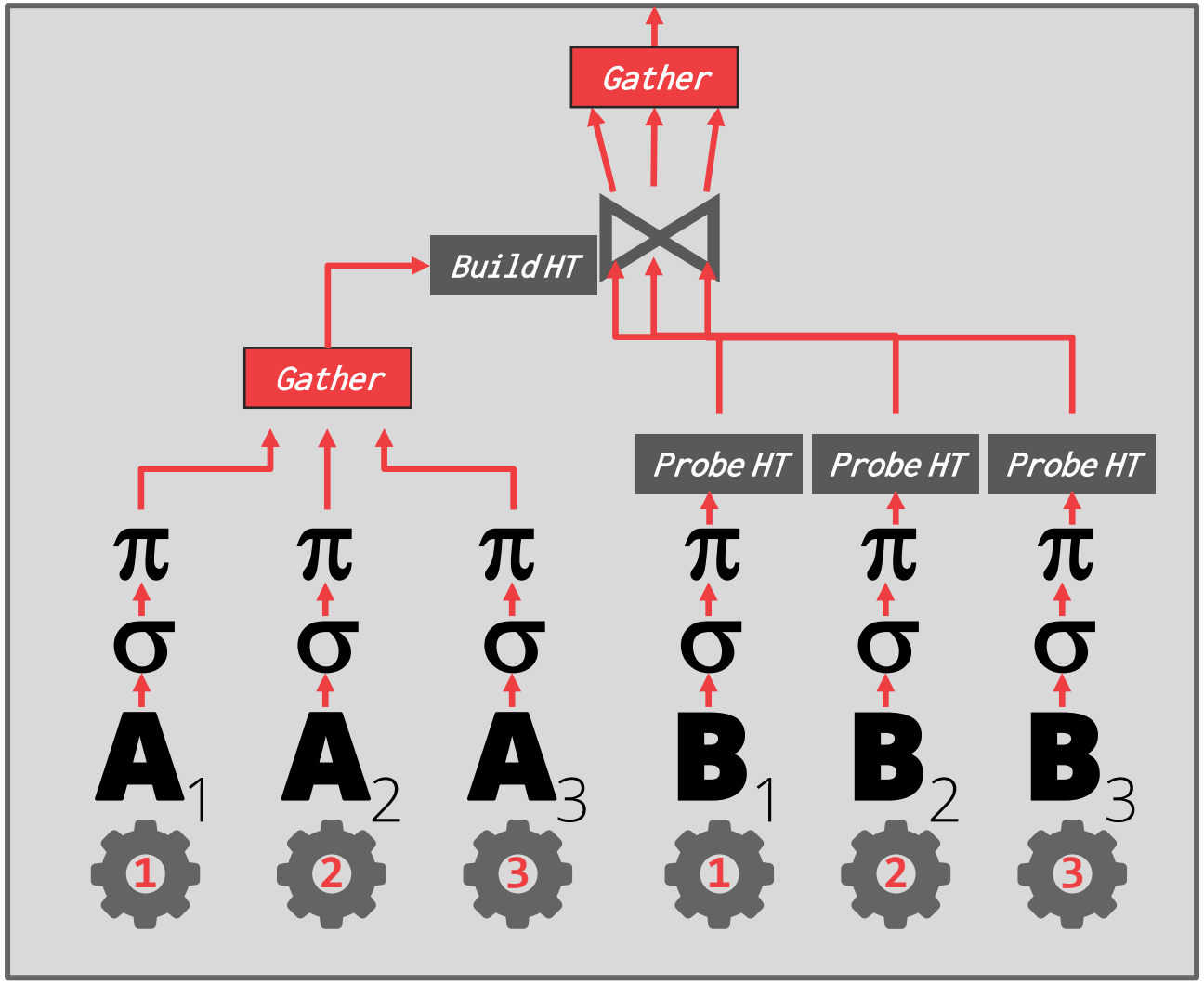
- Data is partitioned across nodes, so we need to consider leveraging the distributed partitioning to accelerate query executions.

- A query can run for a long time -> how to continue query processing under node failures.

Parallel execution for joins

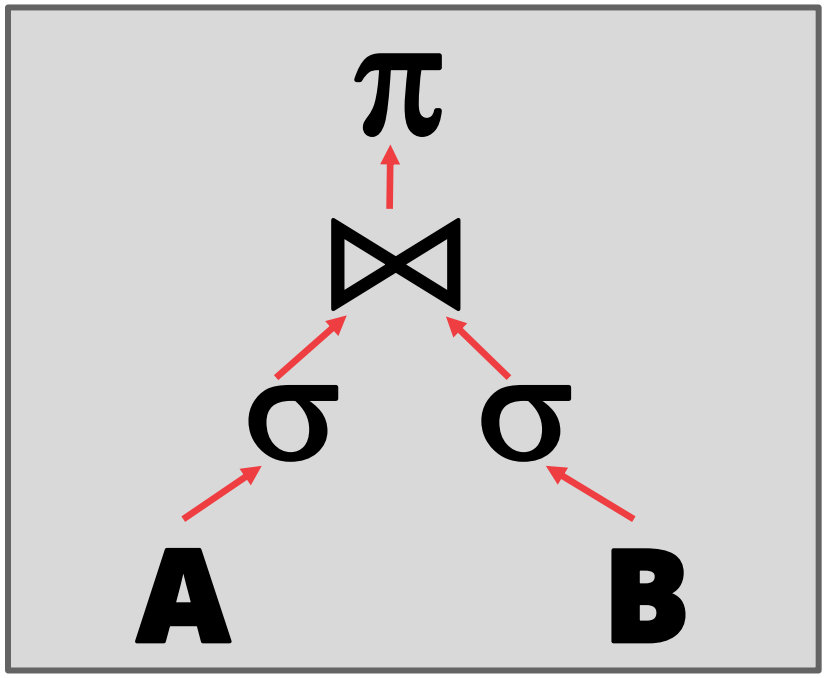
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

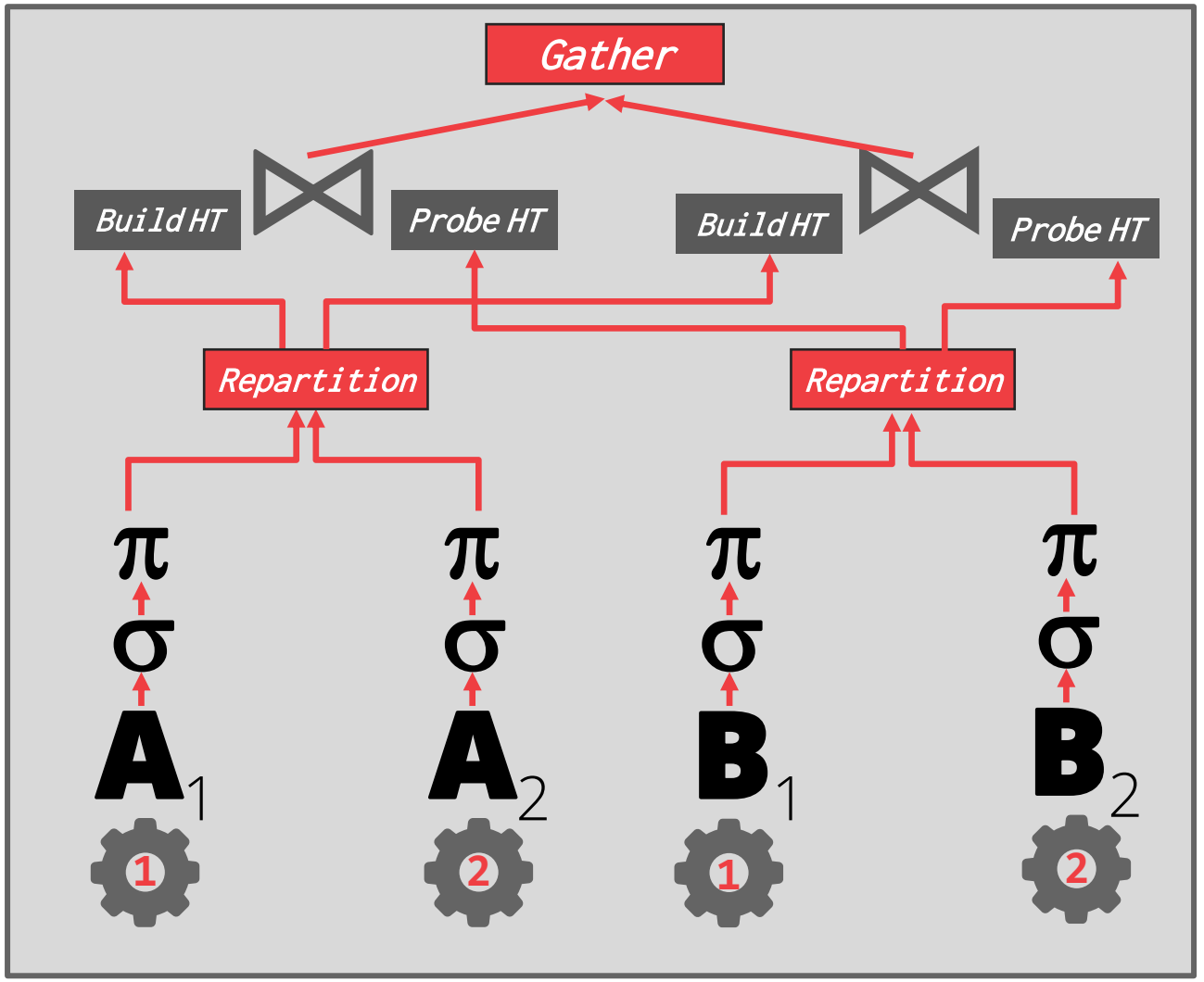




Parallel execution for joins

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```





Distributed join algorithms

- The efficiency of a distributed join depends on the target tables' partitioning schemes.

- One approach is to put entire tables on a single node and then perform the join.
  - This approach loses the parallelism of a distributed DBMS.
  - Costly data transfer over the network.

Distributed join algorithms

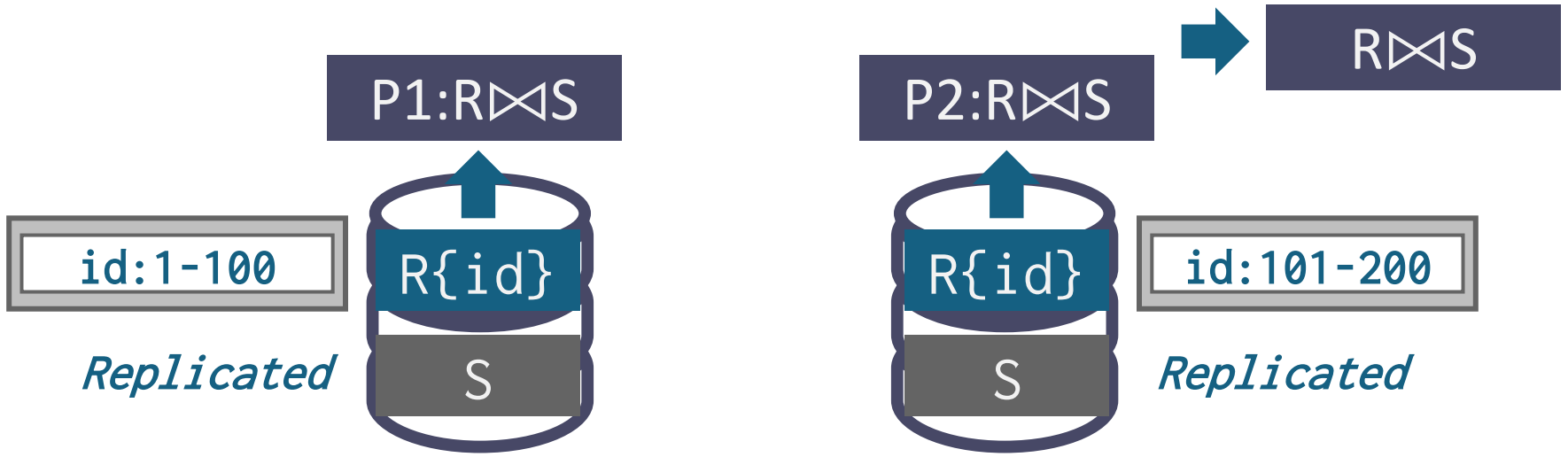
- To join tables R and S, the DBMS needs to get the proper tuples on the same node.

- Once the data is at the node, the DBMS then executes the same join algorithms that we discussed earlier in the semester.
  - Need to produce the correct answer as if all the data is located in a single node system.

Scenario #1

- One table is replicated at every node. Each node joins its local data in parallel and then sends its results to a coordinating node.
  - Which tables to replicate?
  - What is the cost?

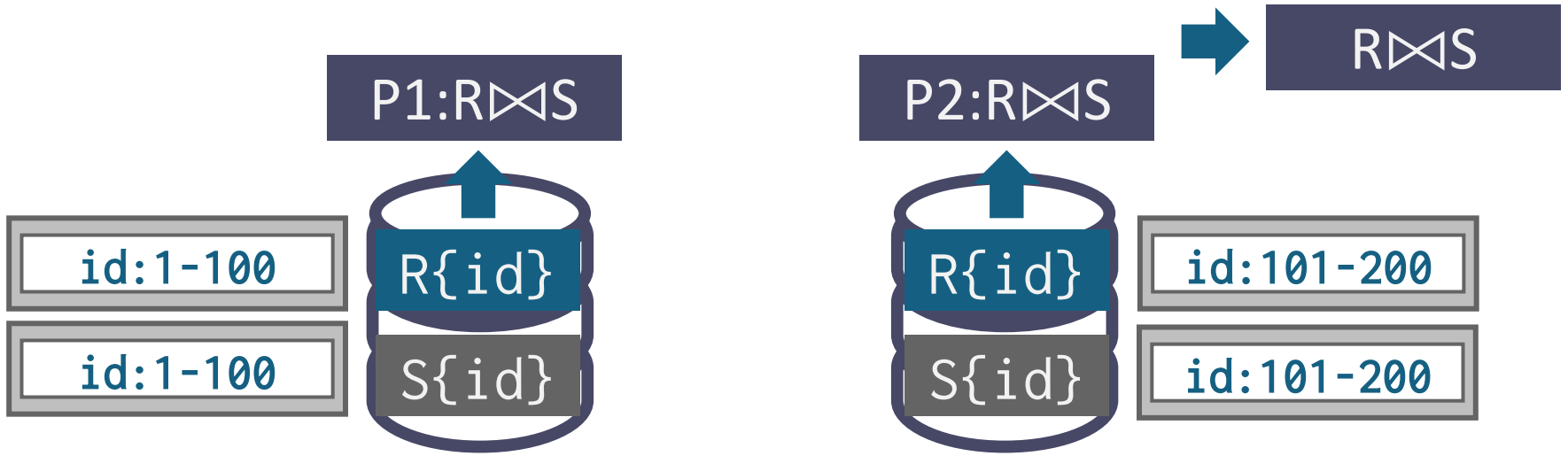
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



Scenario #2

- Tables are partitioned on the join attribute using the same partitioning function.
- Each node performs the join on local data and then sends it to a coordinator node for coalescing.

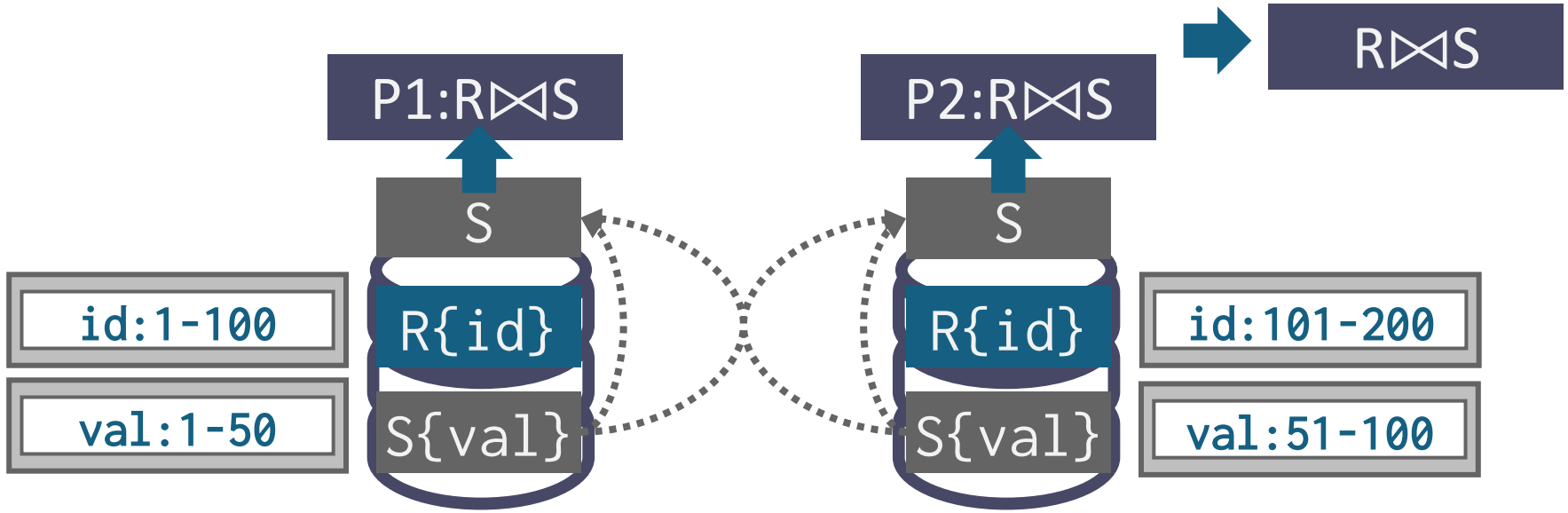
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



Scenario #3

- Both tables are partitioned on different keys. If one of the tables is small, then the DBMS "broadcasts" that table to all nodes.
  - Known as “broadcast join”.

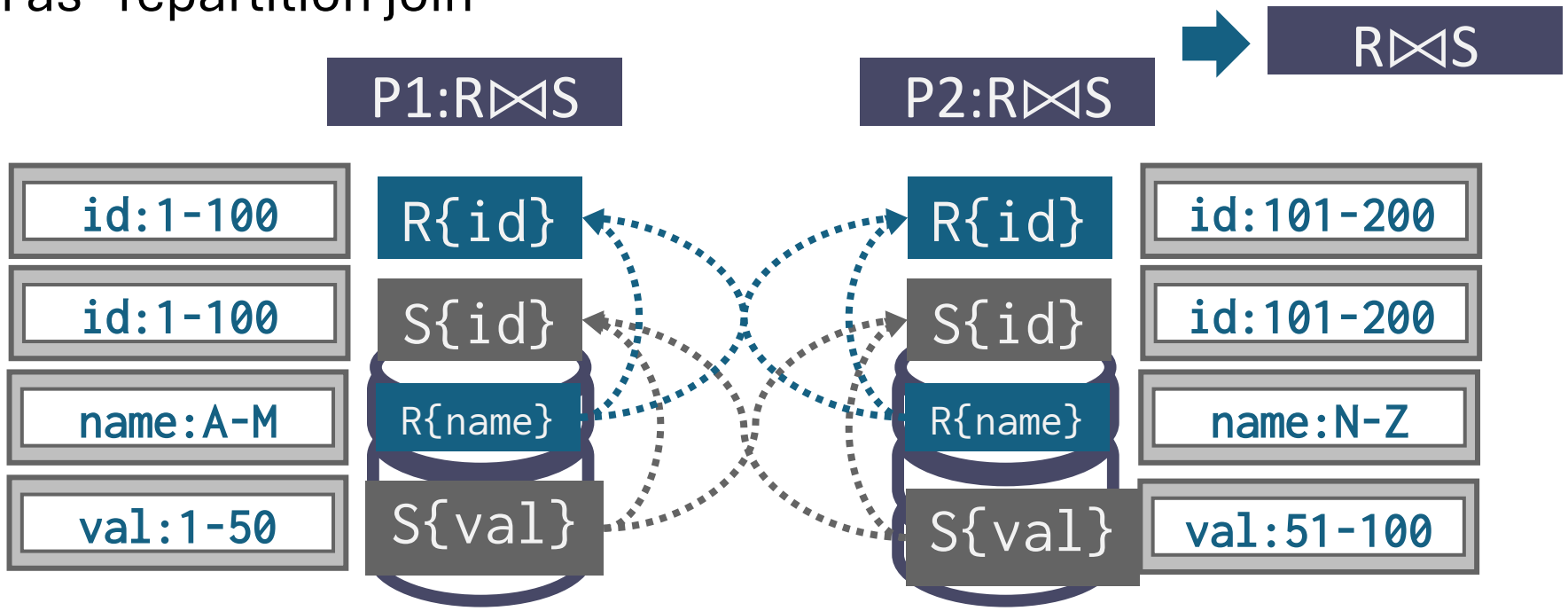
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



Scenario #4

- Both tables are partitioned on different keys. The DBMS copies/re-partitions the tables on the fly across nodes.
  - This repartitioned data is generally deleted when the query is done.
  - Known as “repartition join”

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



Query plan fragments

Approach #1: Physical Operators

- Generate a single query plan and then break it up into partition-specific fragments.
- Most systems implement this approach.

Approach #2: SQL

- Rewrite the original query into partition-specific queries.
- Allows for local optimization at each node.

Query plan fragments

Union the output of each join to produce final result.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

```
SELECT * FROM R JOIN S
  ON R.id = S.id
 WHERE R.id BETWEEN 1 AND 100
```

```
SELECT * FROM R JOIN S
  ON R.id = S.id
 WHERE R.id BETWEEN 101 AND 200
```

```
SELECT * FROM R JOIN S
  ON R.id = S.id
 WHERE R.id BETWEEN 201 AND 300
```

id:1-100

id:101-200

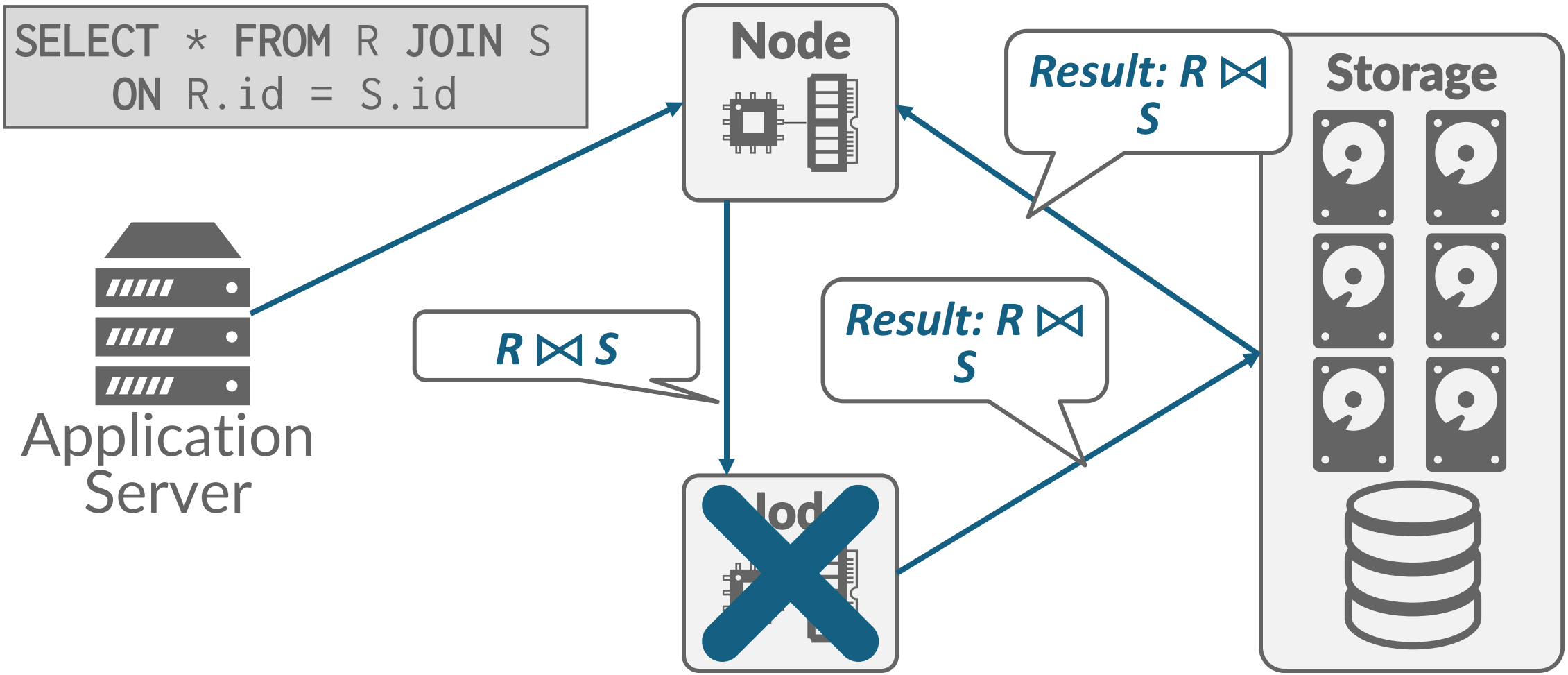
id:201-300

Query fault tolerance

- Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution.
  - If one node fails during query execution, then the whole query fails.

- The DBMS could take a snapshot of the intermediate results for a query during execution to allow it to recover if nodes fail.

Query fault tolerance



MapReduce

- MapReduce is a data processing paradigm proposed by Google.

Motivation

- There is a large volume of raw (unstructured) data to process
  - crawled documents, web request logs, etc.
- They want to build a system that can use hundreds to thousands of machines.
- The system needs to be easy to program so that developers do not need to worry about how to parallelize the data processing.
- Also, the system needs to tolerate node failures when executing a query.

MapReduce

- MapReduce is inspired by the map and reduce primitives of Lisp and many other functional languages.

- Map function:
  - applying the map function to each input record to compute a set of intermediate key/value pairs.
- Reduce function:
  - apply a reduce operation to all the values that share the same key to combine the derived data for the same key together.

MapReduce

- Developers only need to define the map and reduce functions:
  - map (k1,v1) → list(k2,v2)
  - reduce (k2, list(v2))) → list(v3)

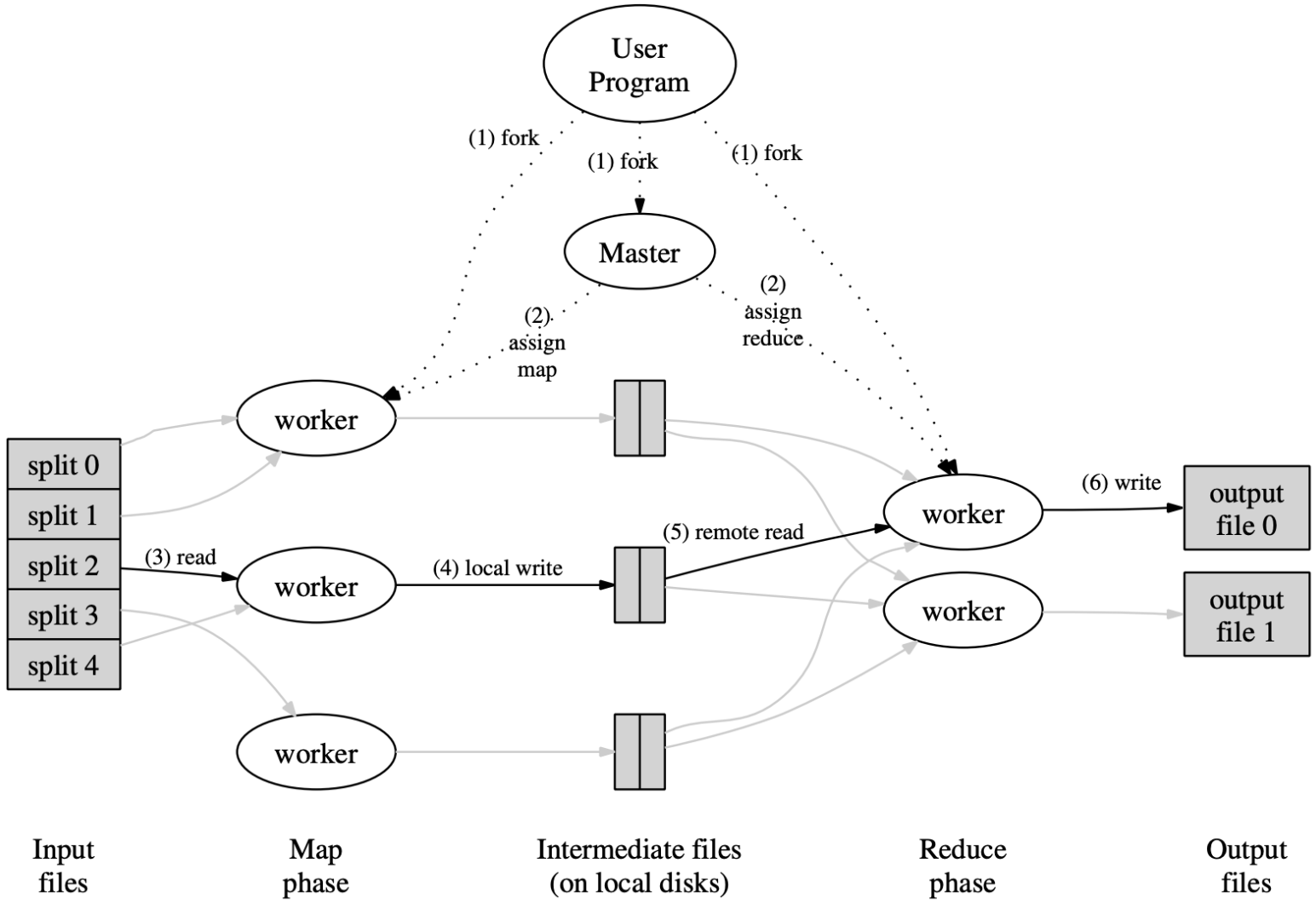
- The system will handle
  - Data distribution
  - Parallel execution
  - Fault tolerance

Example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Execution Overview



Execution Overview

- A master process will manage the execution of mappers and reducers
  - Scheduling
  - Managing metadata: where is the output of the mapper
  - Fault tolerance

Execution Overview

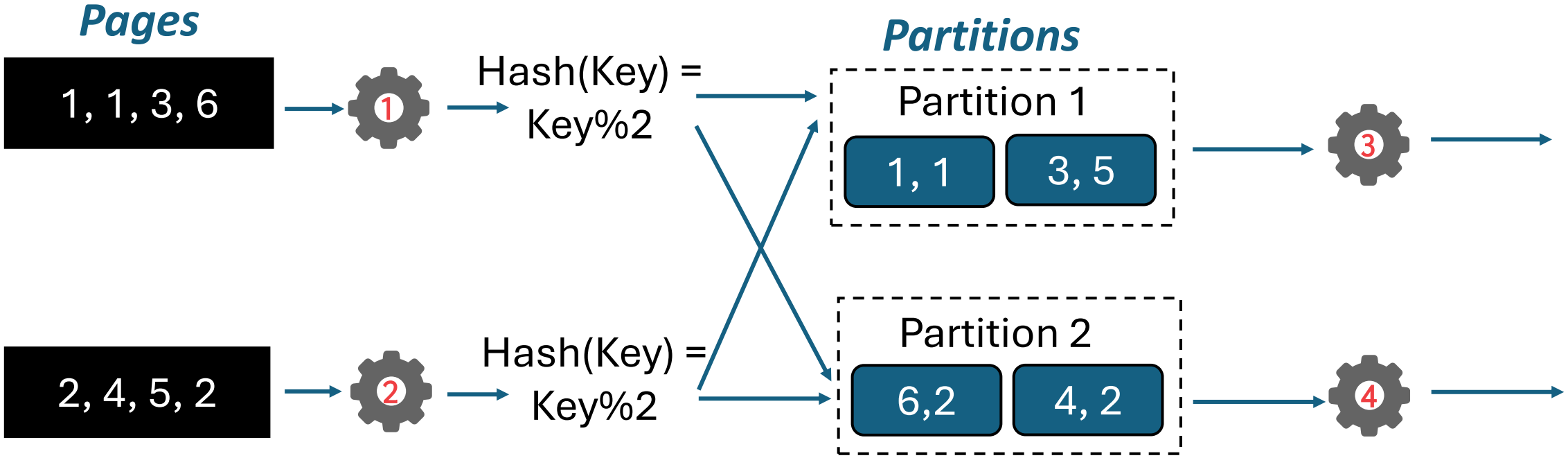
- Map Phase:
  - Input data is partitioned into M splits.
    - Each split is 64 MB – 256 MB.
  - Each map function will process one split and map functions are spread over multiple machines.
  - The output of a map function is automatically partitioned based on the reduce key (k2) and stored on the local disk.

Execution Overview

- Reduce Phase:
  - A reducer will copy a partition of data from each mapper.
  - After it gets the partitions from all mappers, it sorts the data by the reduce key k2 to generate a list of (k2, list(v2)).
  - The reduce function is called for each (k2, list(v2)).

Parallel execution of aggregation

- MapReduce is very similar to the parallel execution of aggregation
  - The difference is that both map/reduce are UDFs, providing the flexibility of processing raw (unstructured) data.



Fault tolerance

- The master process pings every worker periodically to detect worker failures.
- Any map tasks executed by a failed worker are re-executed
  - This is because the local disk for the failed worker is not inaccessible.
- Any reduce tasks in progress are re-executed
- Completed reduce tasks do not need to be executed
  - The output of a reducer is on a distributed file system.

Data locality

- The input data is stored in a distributed file system
  - Each split is replicated (typically 3 copies)

- It is preferable to schedule a mapper to a node that has a copy it needs to process.
- If not, schedule the mapper closer to that copy
  - Within a rack or sharing a switch
- This is to reduce network bandwidth usage.

Stragglers

- With a large number of mappers and reducers, there will be some of them executing very slowly, called stragglers
  - Scheduling problems
  - Slow/bad local disks

- Solution: backup tasks
  - The master process will execute the same task on multiple workers.
  - If one of them finishes, the task is complete.

Data file formats

- Most DBMSs use a proprietary on-disk binary file format for their databases.
- The only way to share data between systems is to convert data into a common text-based format
  - Examples: CSV, JSON, XML
- There are new open-source binary file formats that make it easier to access data across systems.

Data file formats

Apache Parquet

- Compressed columnar storage from Cloudera/Twitter

Apache ORC

- Compressed columnar storage from Apache Hive.

Apache CarbonData

- Compressed columnar storage with indexes from Huawei.

Apache Iceberg

- Flexible data format that supports schema evolution from Netflix.

HDF5

- Multi-dimensional arrays for scientific workloads.

Apache Arrow

- In-memory compressed columnar storage from Pandas/Dremio.

Recall: Parquet

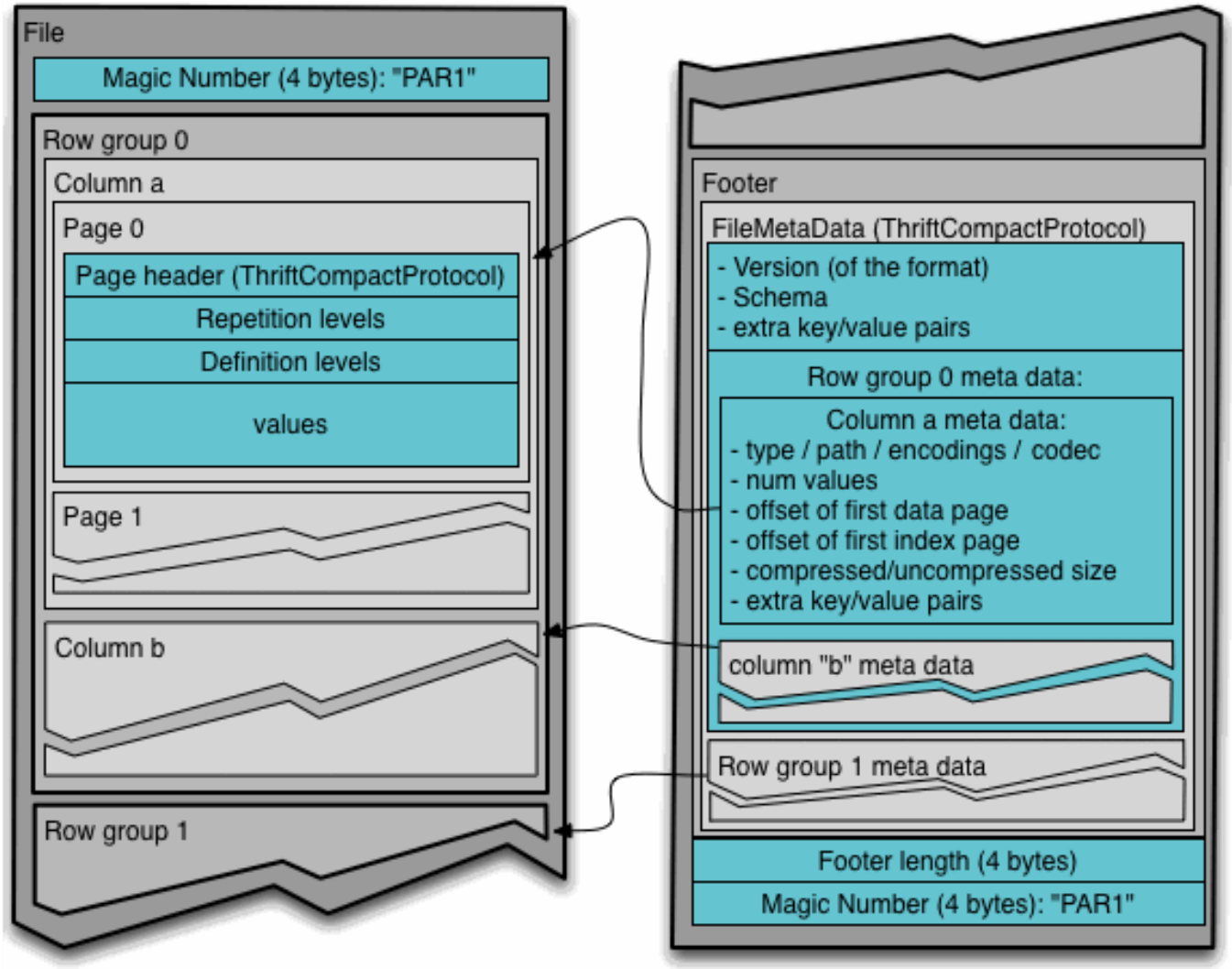
- Apache Parquet is an open-source, column-oriented data file format.
- It provides high-performance compression and encoding schemes to handle complex data in bulk.
- A parquet file consists of one or more row groups.
  - A row group is a partition of rows and includes a column chunk for each column in the dataset.
  - A column chunk is guaranteed to be contiguous in the file and divided up into pages.
    - A page is conceptually an indivisible unit (in terms of compression and encoding).
    - There can be multiple page types that are interleaved in a column chunk.

Parquet file format

- There are N columns in this table, split into M row groups.
- The file metadata contains the locations of all the column chunk start locations.
- File metadata is written after the data to allow for single-pass writing.
- Readers first read the file metadata to find all the column chunks they are interested in.
- The column chunks should then be read sequentially.

```
4-byte magic number "PAR1"
<Column 1 Chunk 1>
<Column 2 Chunk 1>
...
<Column N Chunk 1>
<Column 1 Chunk 2>
<Column 2 Chunk 2>
...
<Column N Chunk 2>
...
<Column 1 Chunk M>
<Column 2 Chunk M>
...
<Column N Chunk M>
File Metadata
4-byte length in bytes of file metadata (little endian)
4-byte magic number "PAR1"
```

File metadata



Today’s agenda

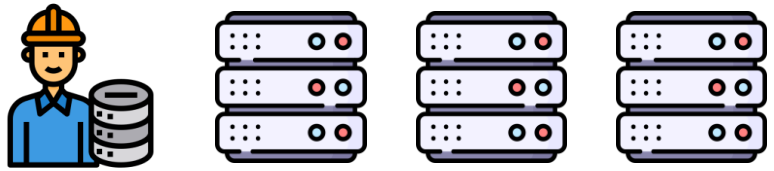
- Current practice in distributed OLAP

- Cost-intelligent data analytics in the cloud

Recall cloud databases

Traditional

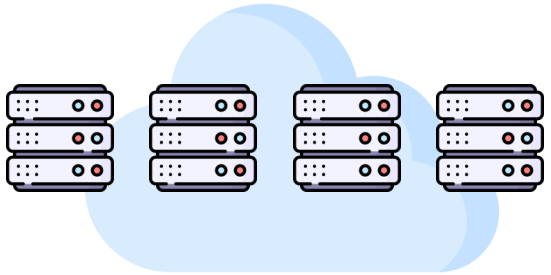
$$$



$$$

Cloud-Native

$ pay-as-you-go



• • • Elasticity

Database optimization in the cloud era

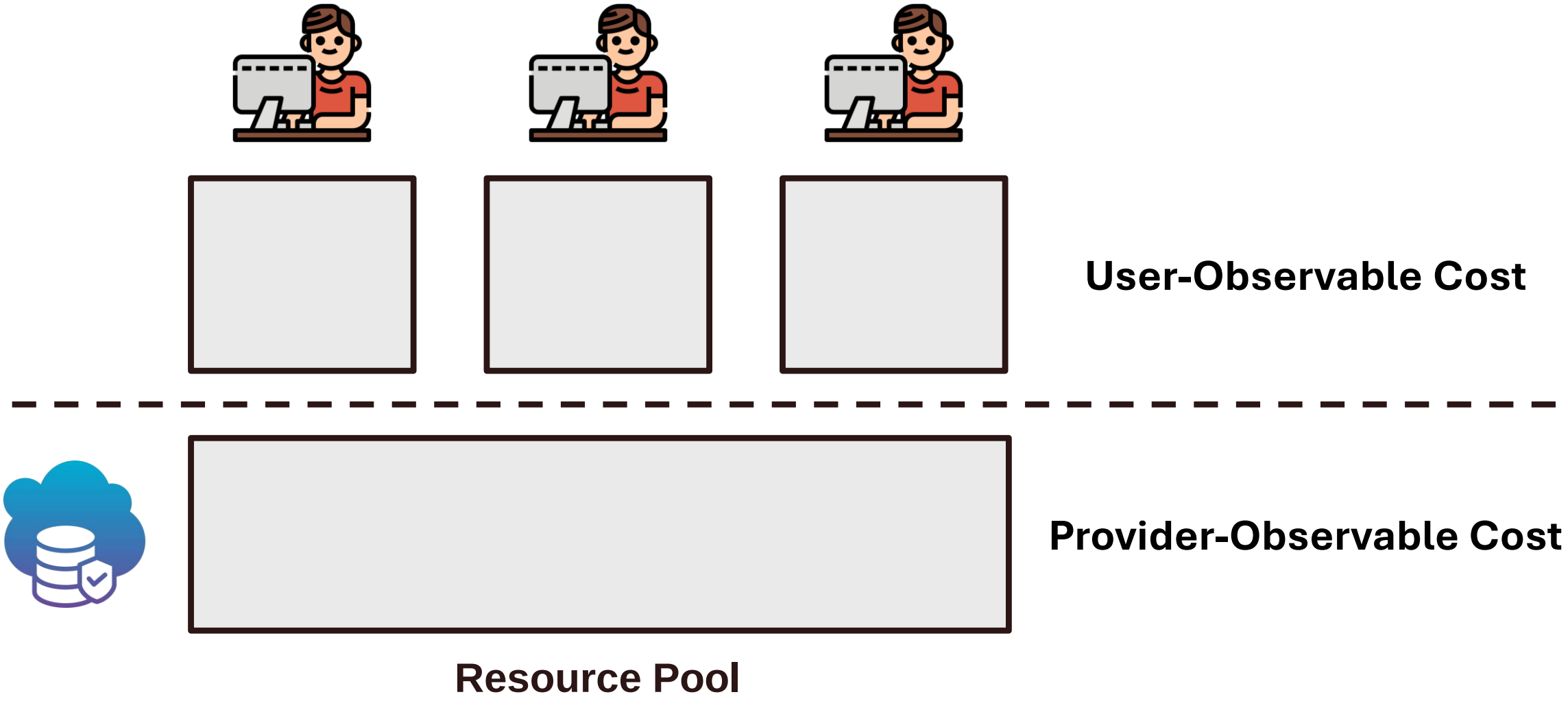
	User Profit	Utility	$ Cost
	Π	$U(p)$	$-C$
Traditional		$p \uparrow$	$C_{sunk} + C_{\epsilon}$
Cloud-Native		p	ΔC

Fix, Large

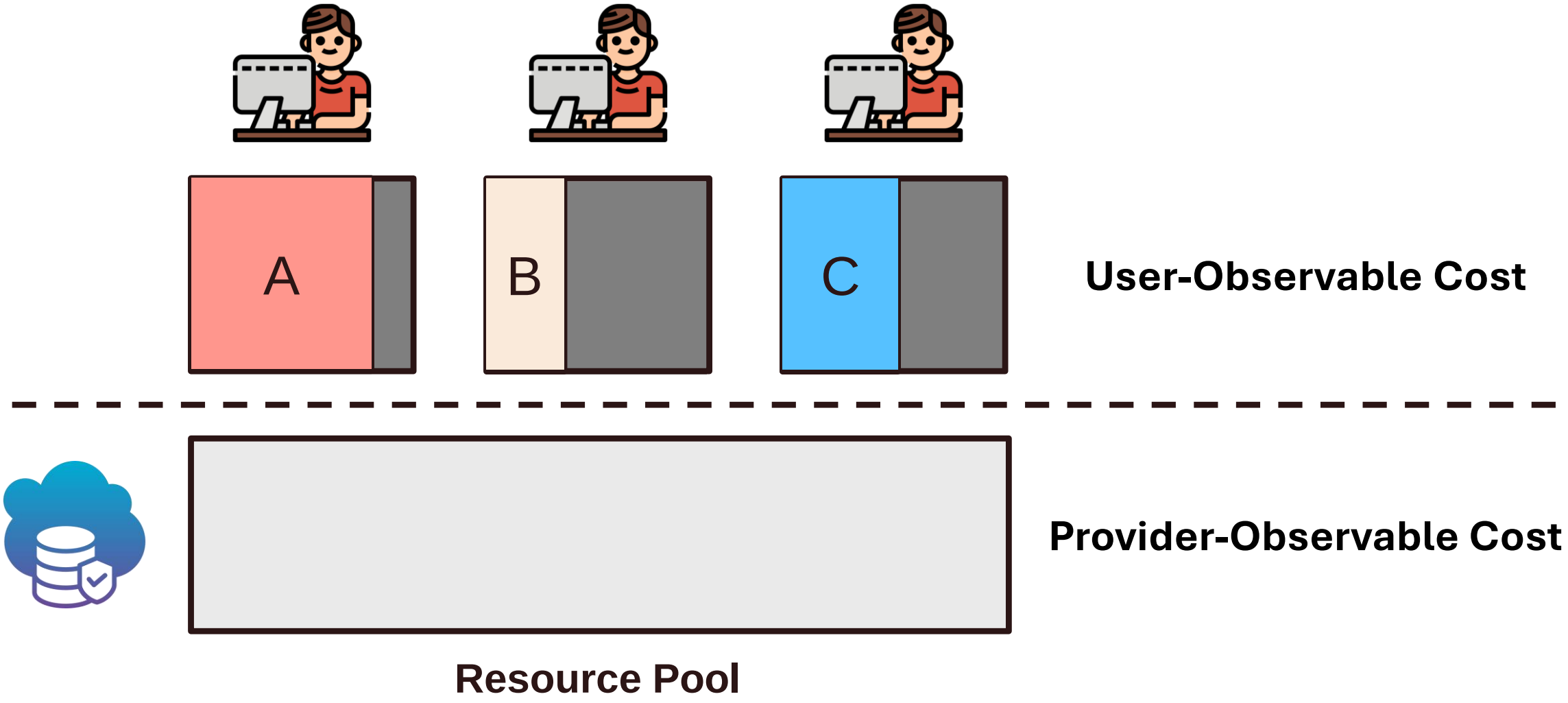
Enabled by **Elasticity**

Bi-Objective Optimization

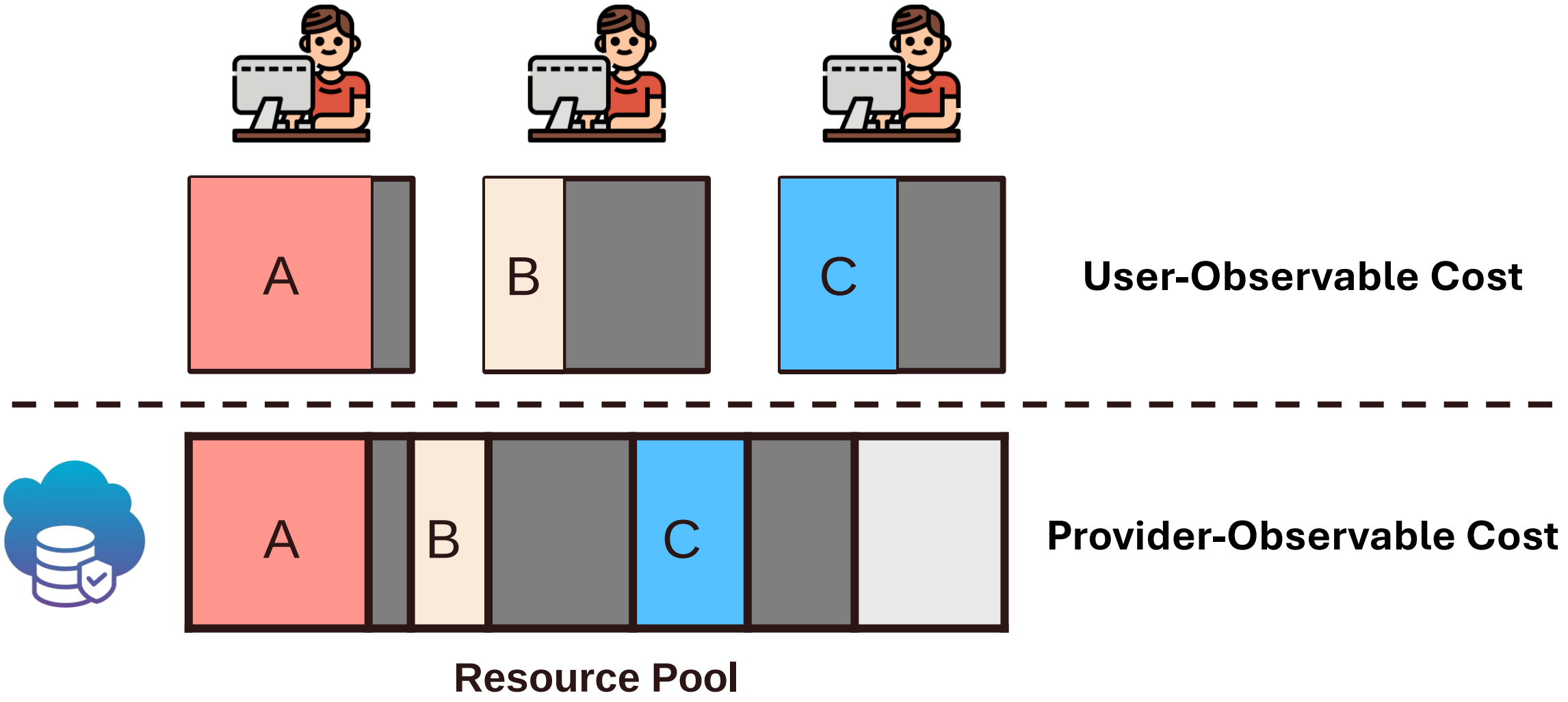
Cost from different perspectives



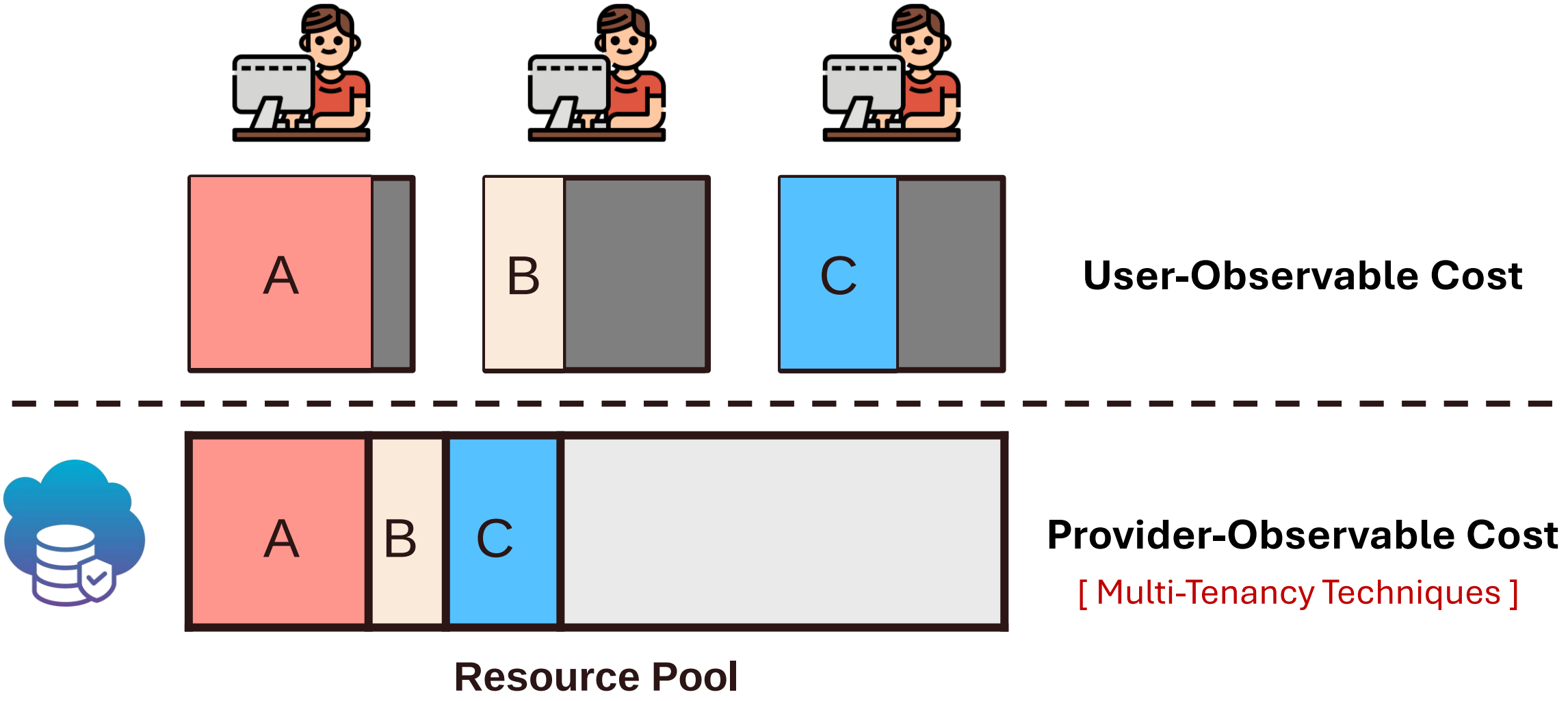
Cost from different perspectives



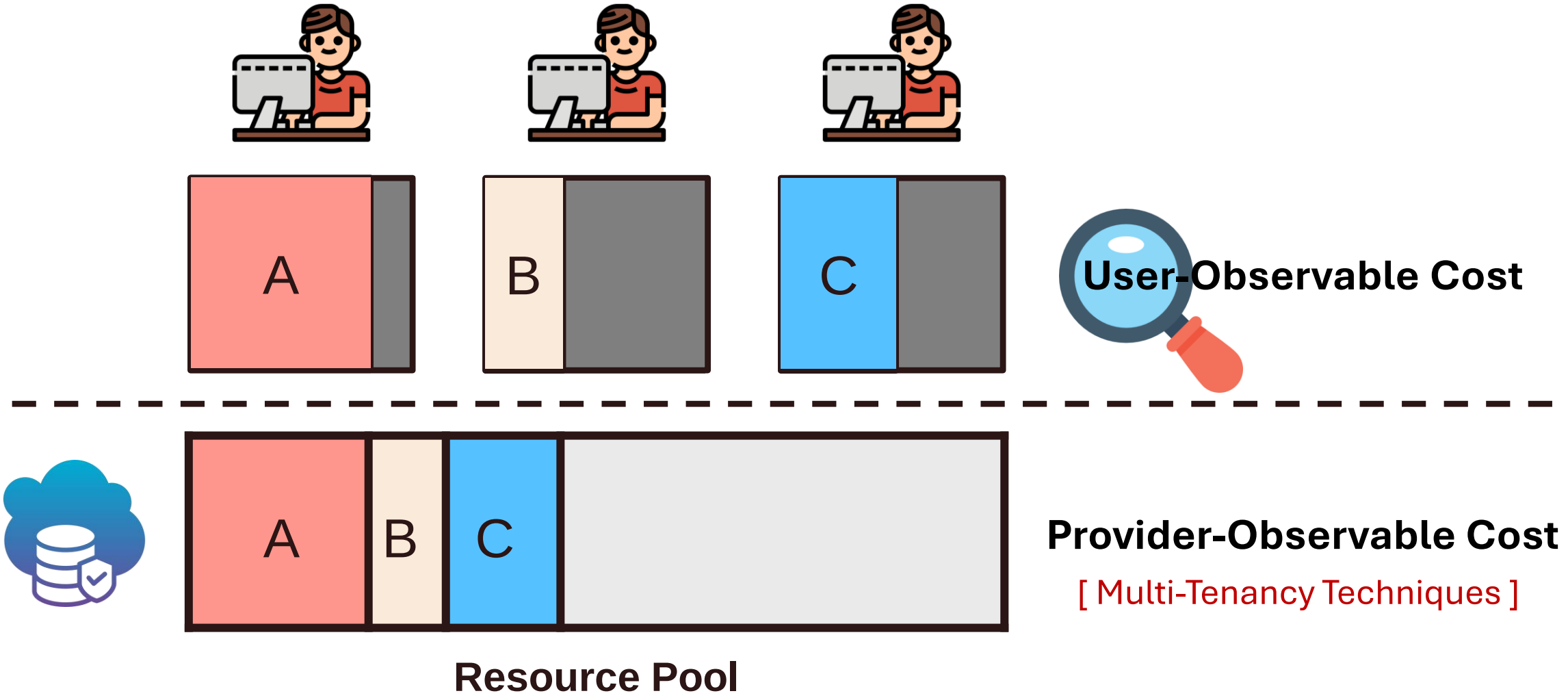
Cost from different perspectives



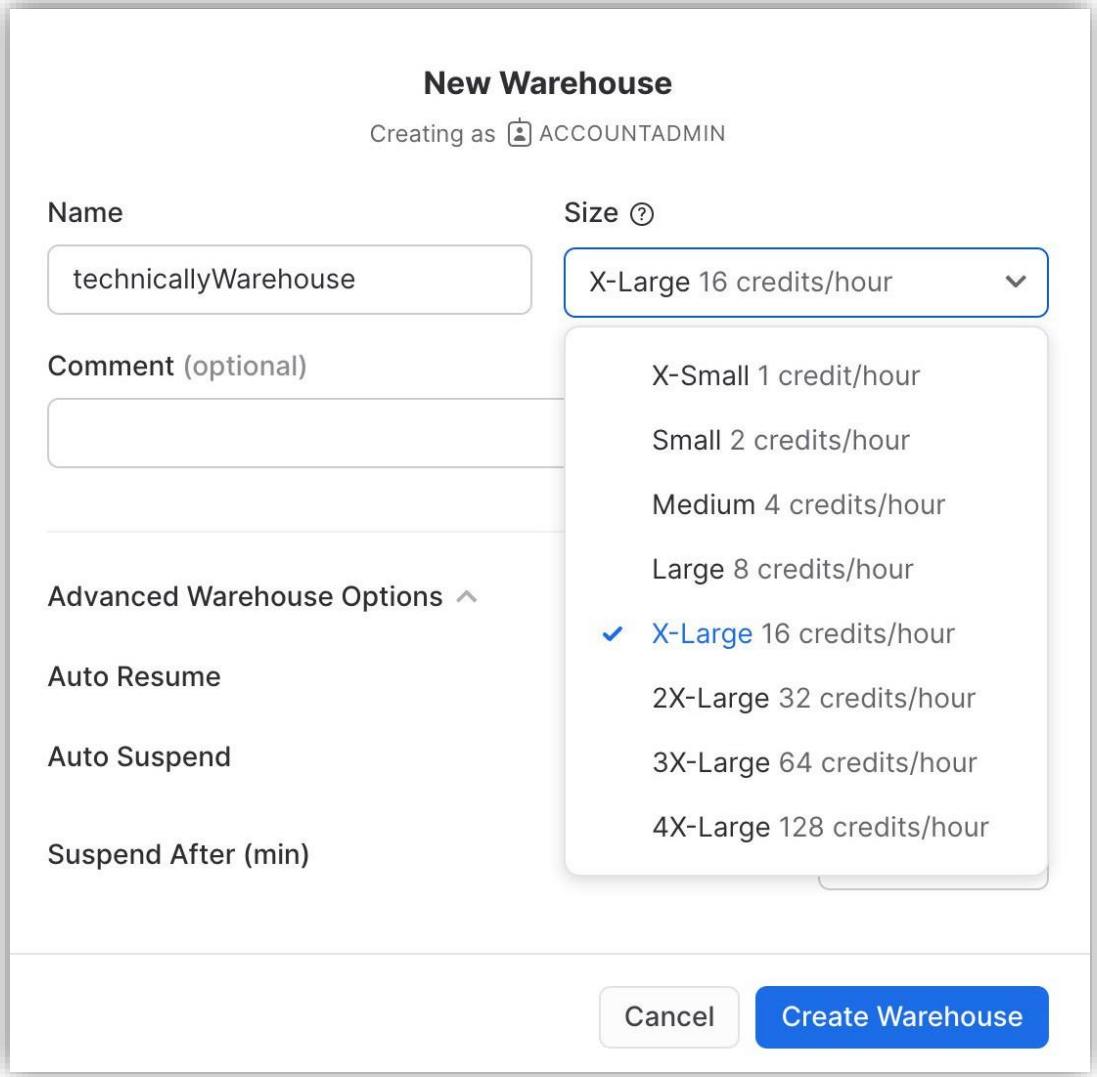
Cost from different perspectives



Cost from different perspectives



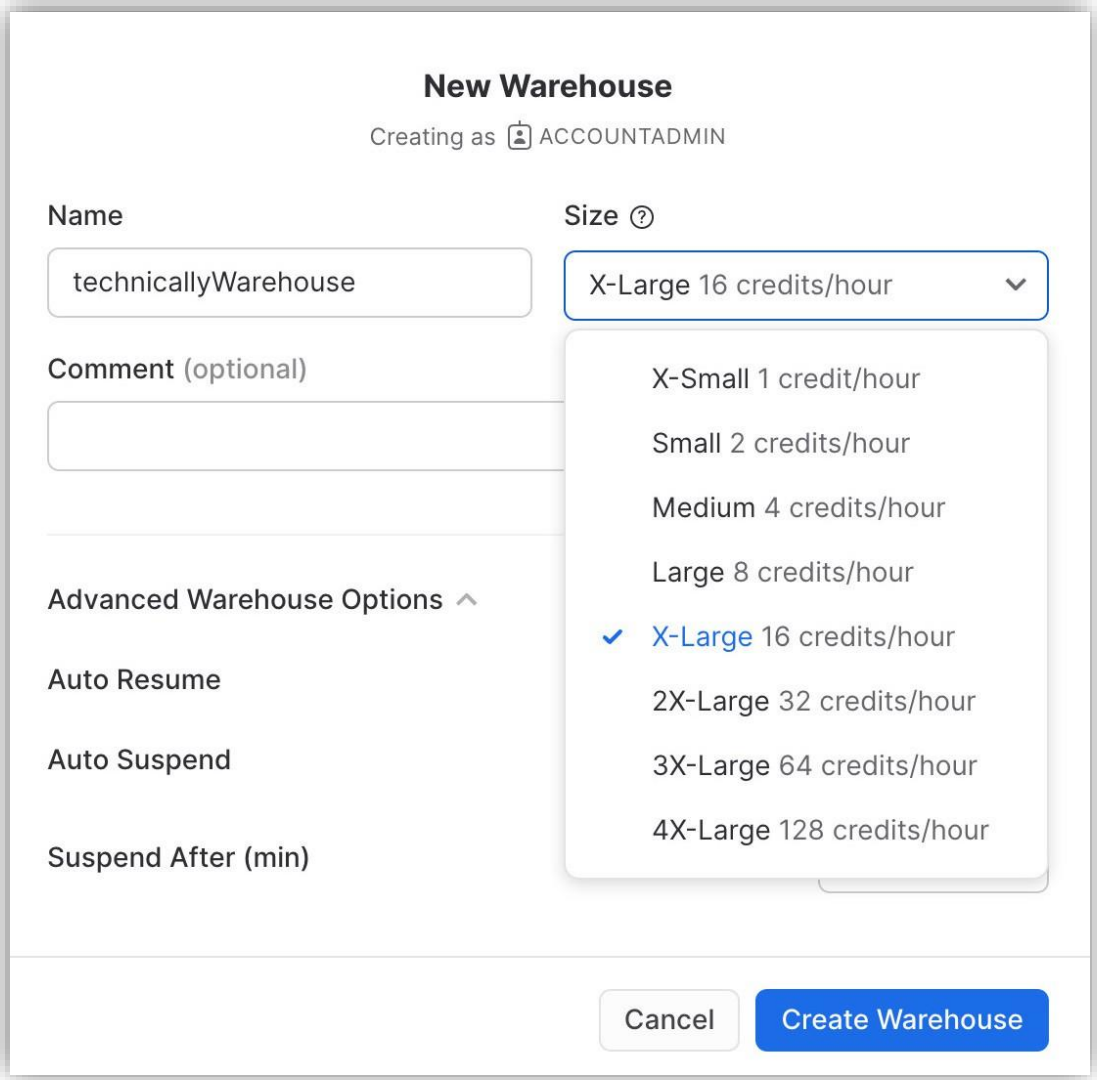
Cost control is still difficult



➔ **Users tend to over-provision**

➔ **Fixed cluster size over the entire workload**

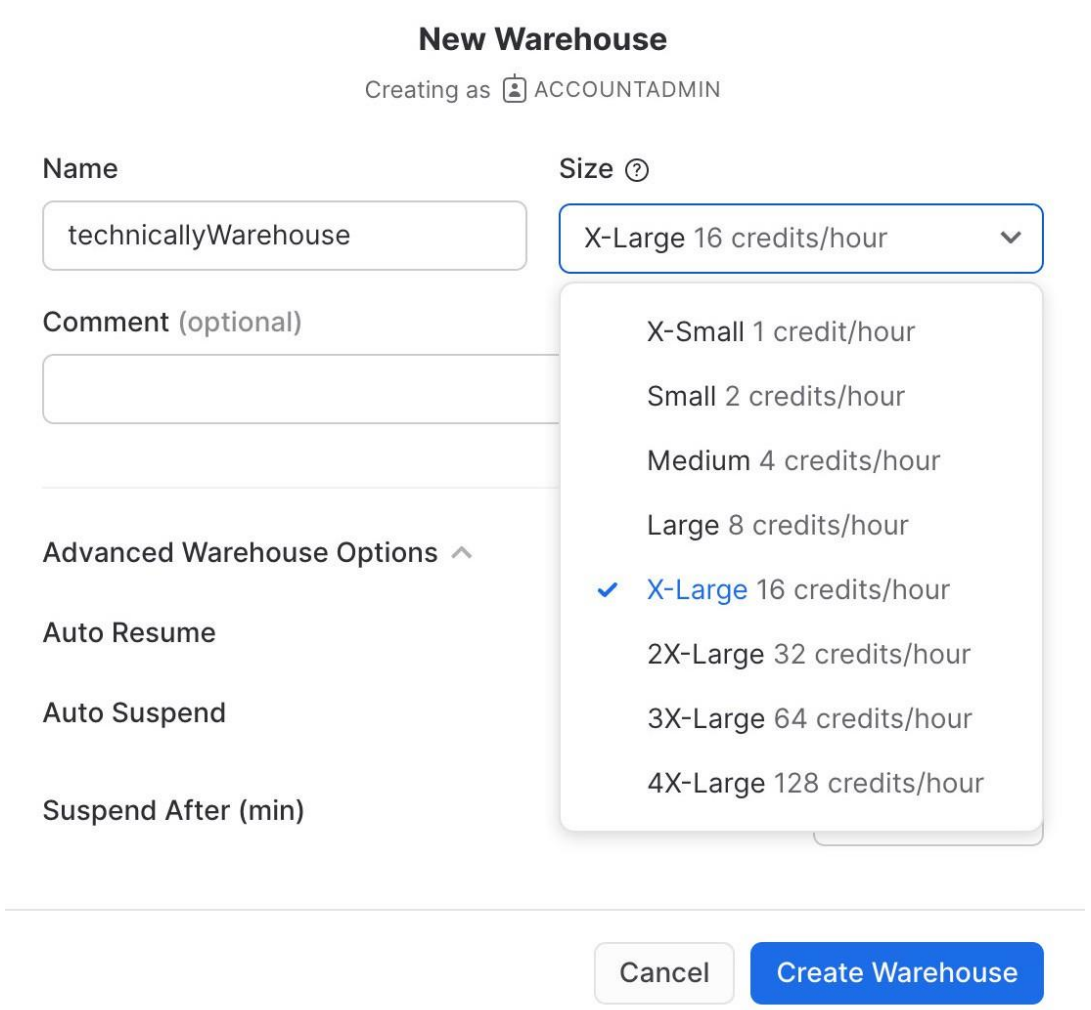
Cost control is still difficult



- ➔ **Users tend to over-provision**
- ➔ **Fixed cluster size over the entire workload**

Resource Waste!

Cost control is still difficult



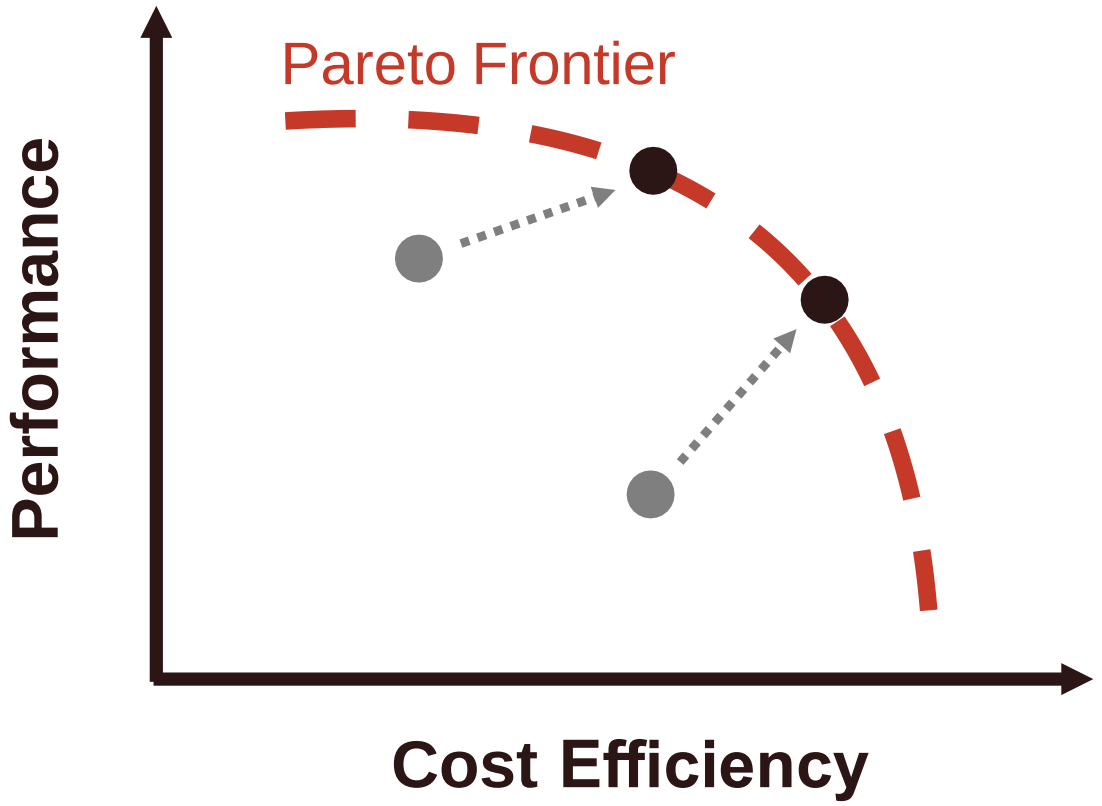
- Build Indexes
- Build Materialized Views
- Re-partition Data
- Re-train a Learned Module

DBA

$$$

Cost intelligence

The system's ability to **self-adapt** to stay at the **Pareto Frontier** in the performance- cost trade-off under different workloads and user constraints.



An Idea UI

Workload 

Time: 10s  10min

Cost: $2  $0.1

- Build Indexes
- Build Materialized Views
- Re-partition Data
- Re-train a Learned Module

DBA

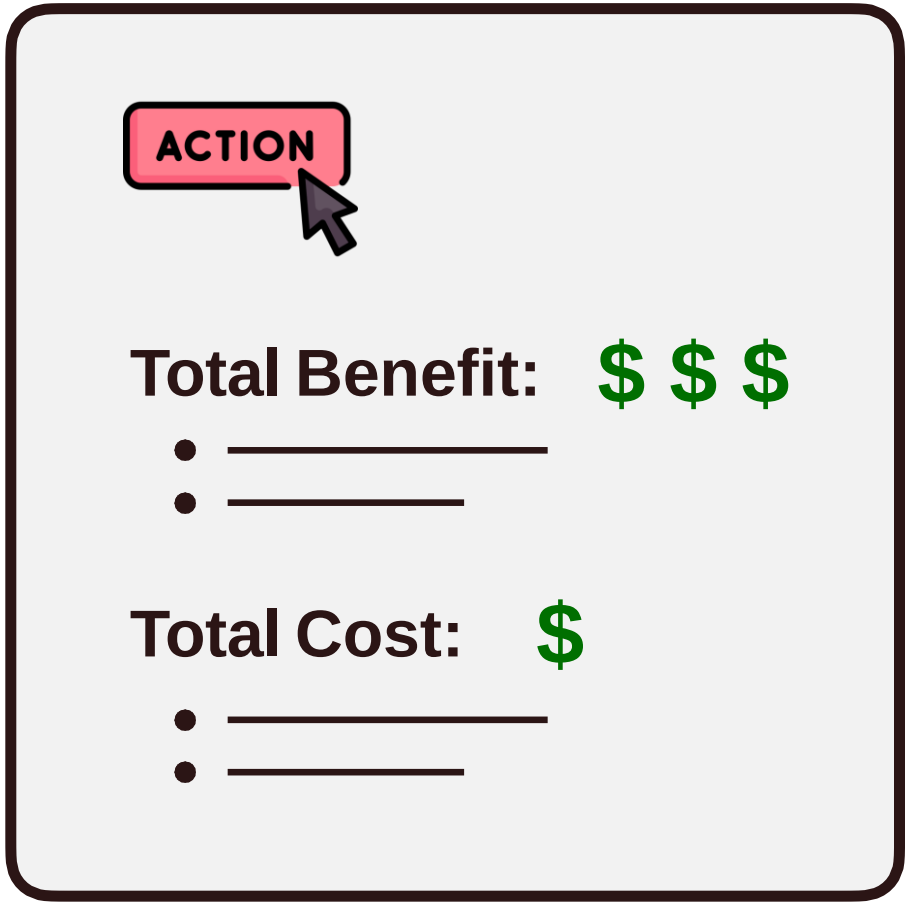
$$$

An Idea UI

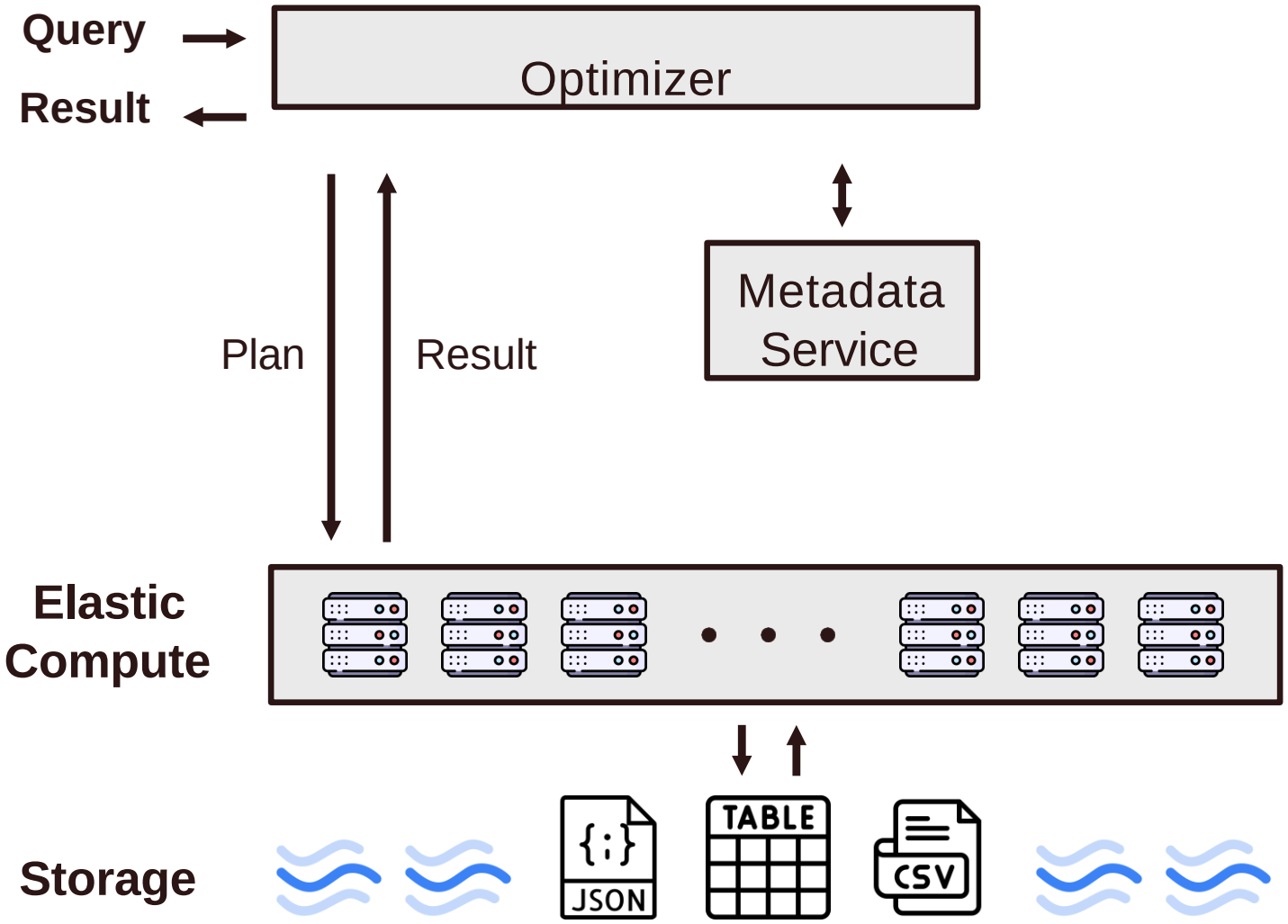
Workload 

Time: 10s ——— 10min

Cost: $2 ——— $0.1



Base system architecture



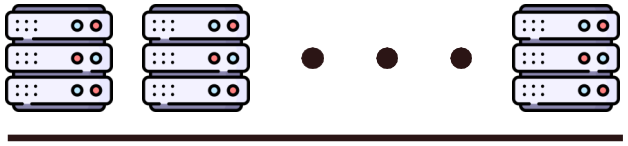
Automatic resource deployment

Workload

Config 1

× 100 min

Config 2



100 servers

× 1 min

Same $ Cost **100x performance boost!**

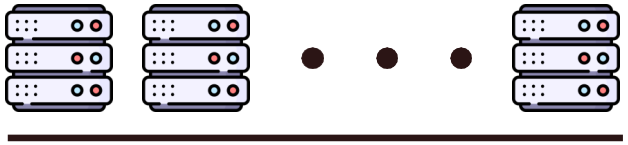
Automatic resource deployment

Workload

Config 1

100 min

Config 2



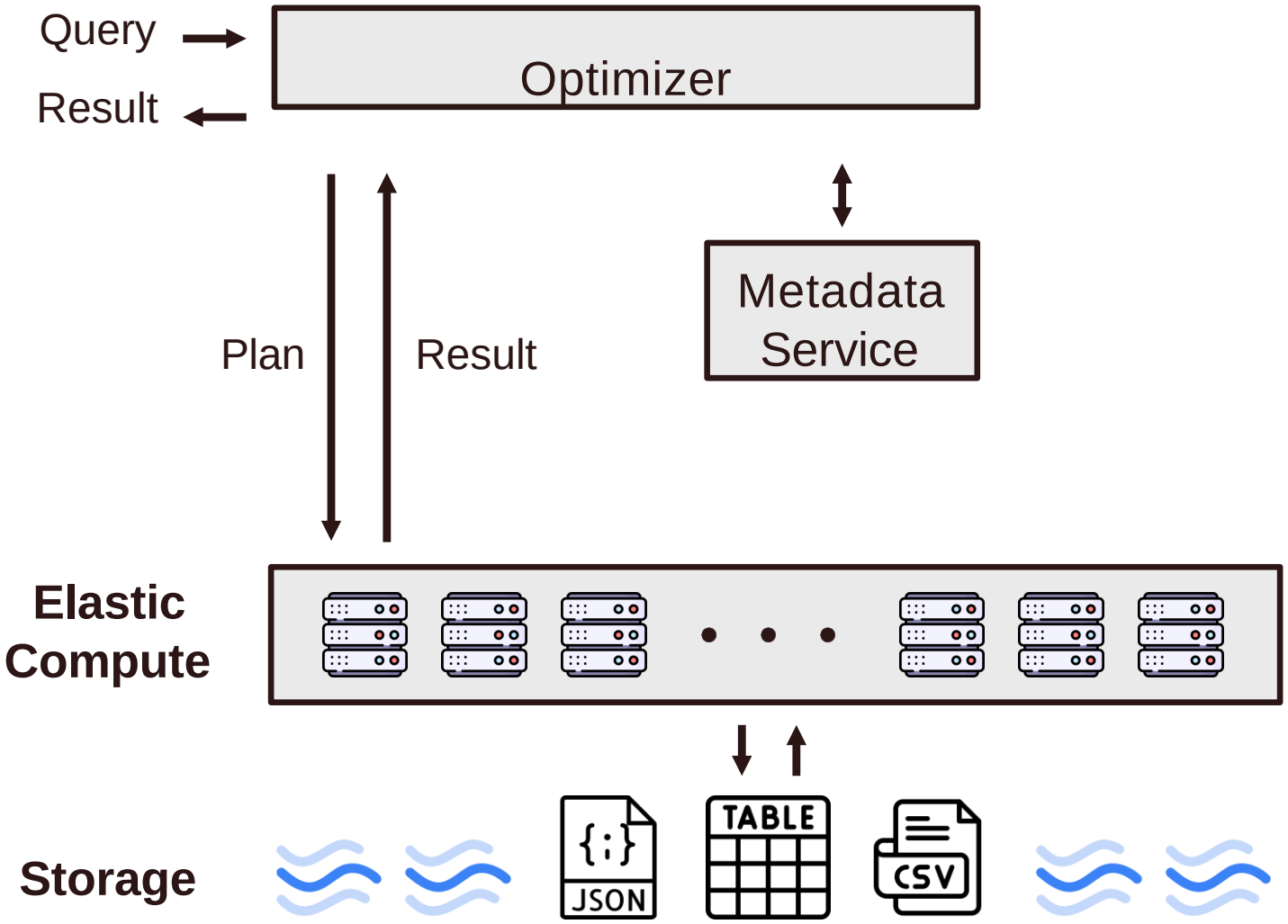
100 servers

100 min

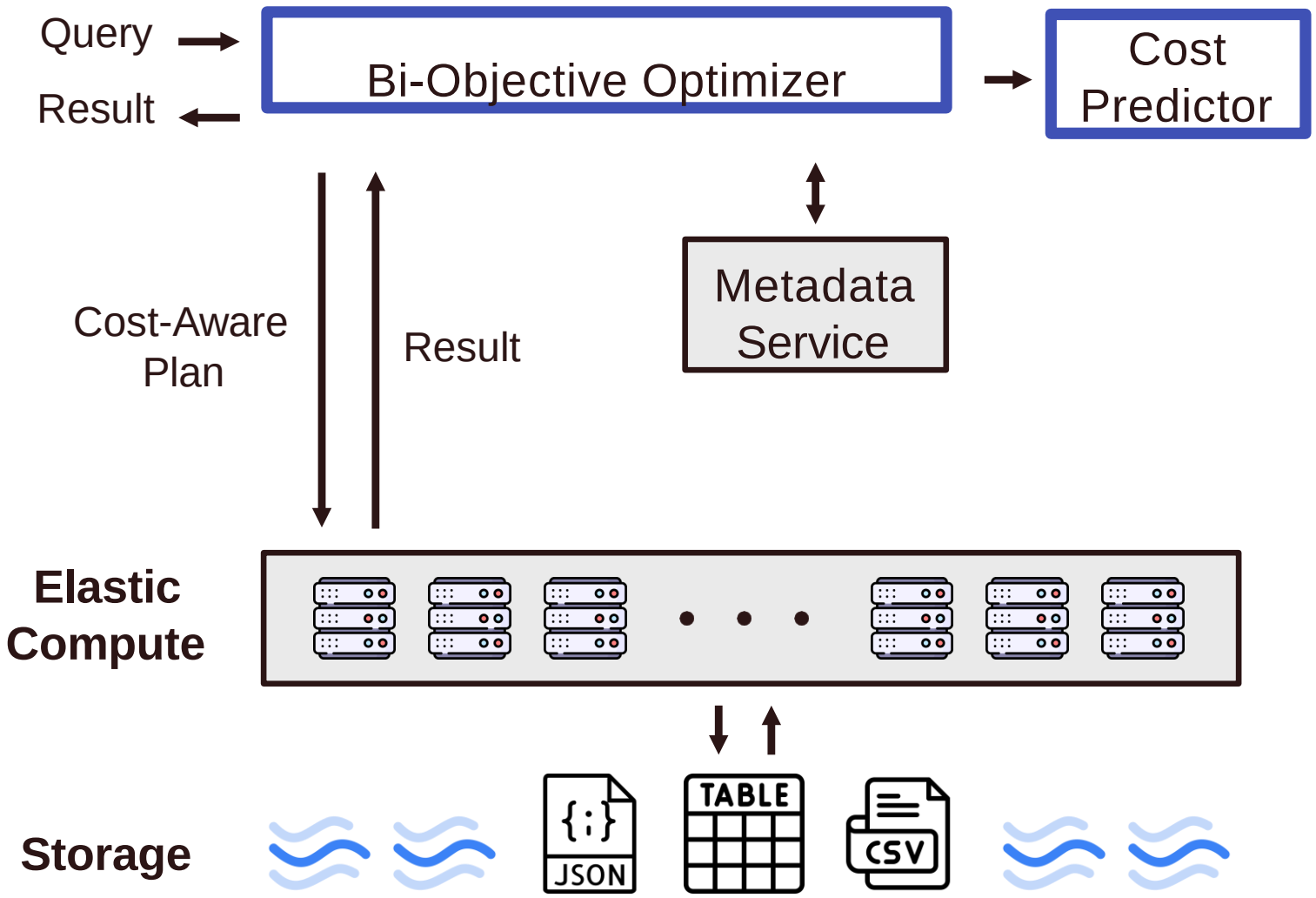
100x $ Cost

Same performance

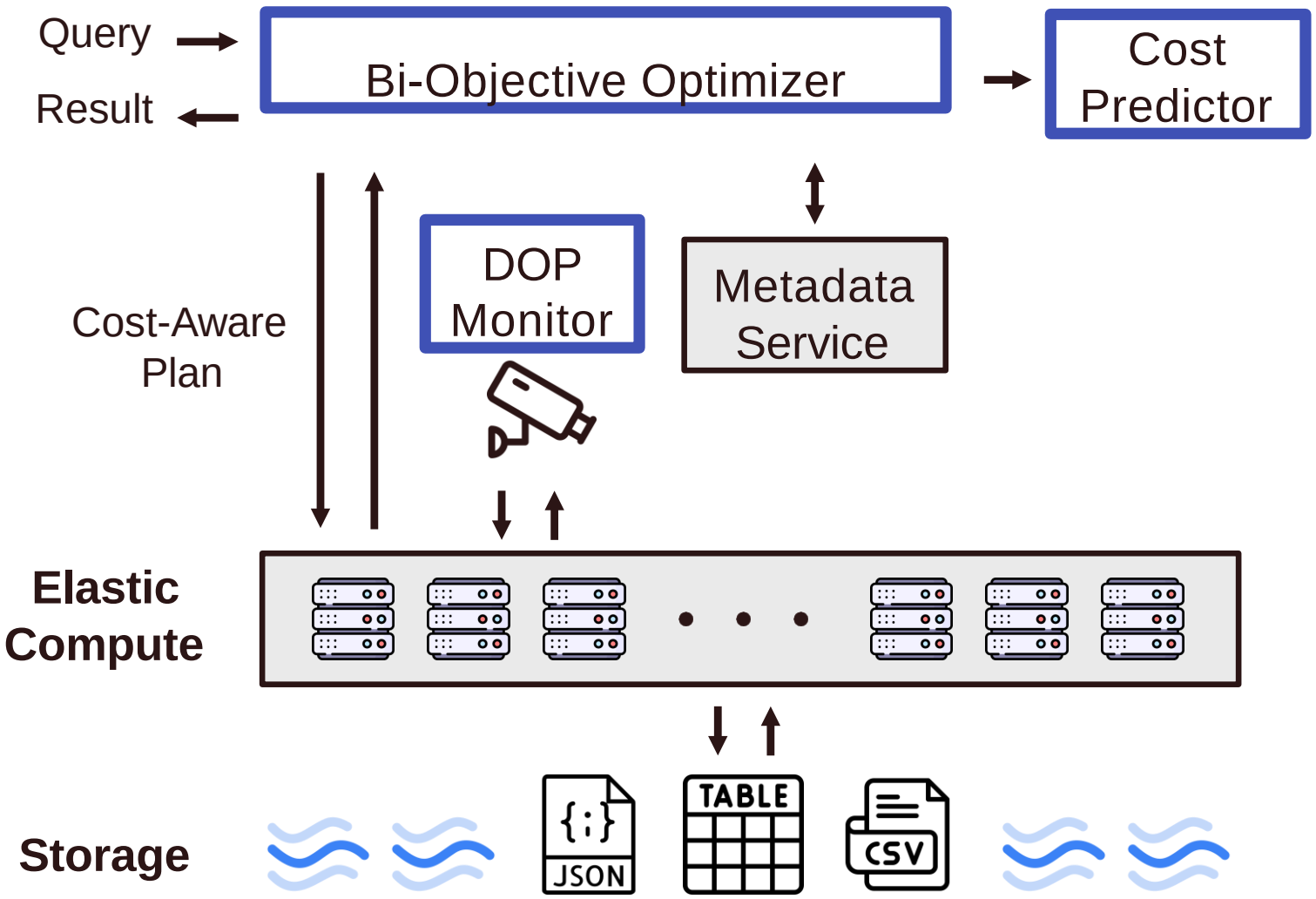
System architecture



System architecture

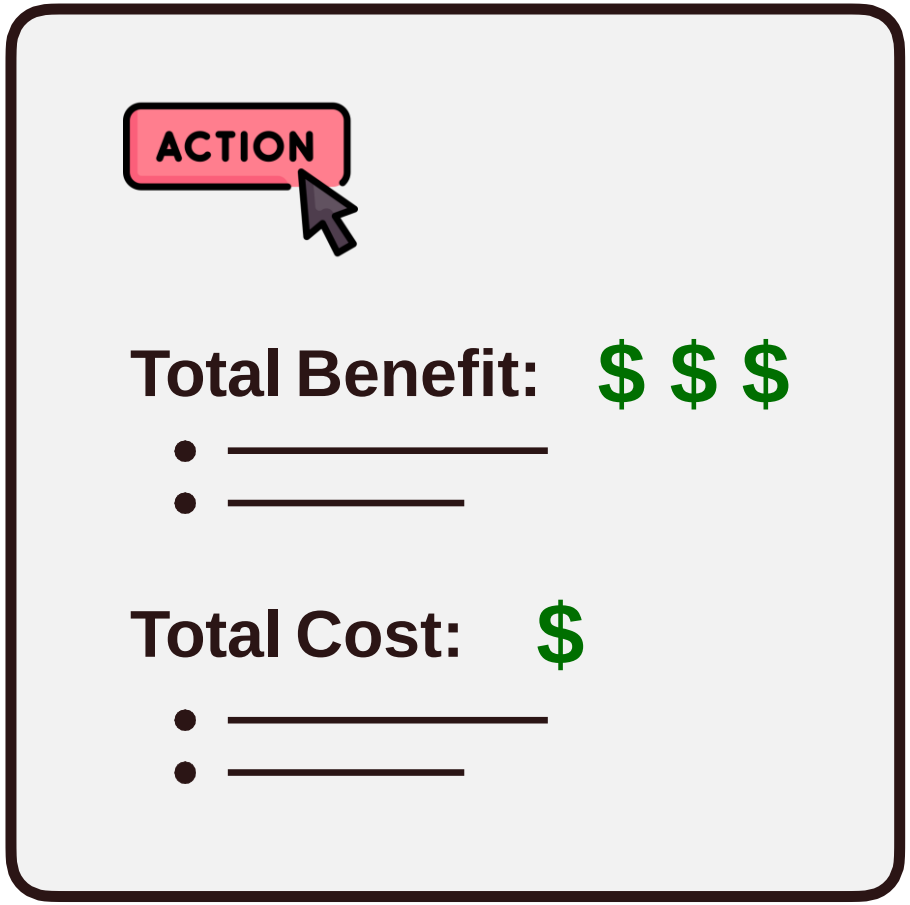


System architecture

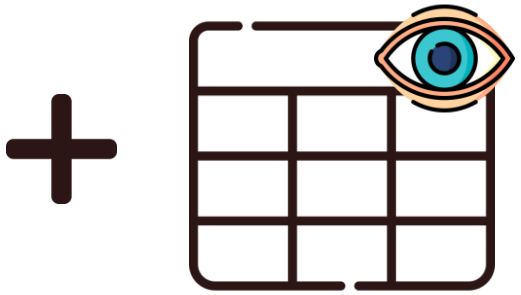


Cost-oriented database auto-tuning

- Build Indexes
- Build Materialized Views
- Re-partition Data
- Re-train a Learned Module



Database tuning under fixed resources



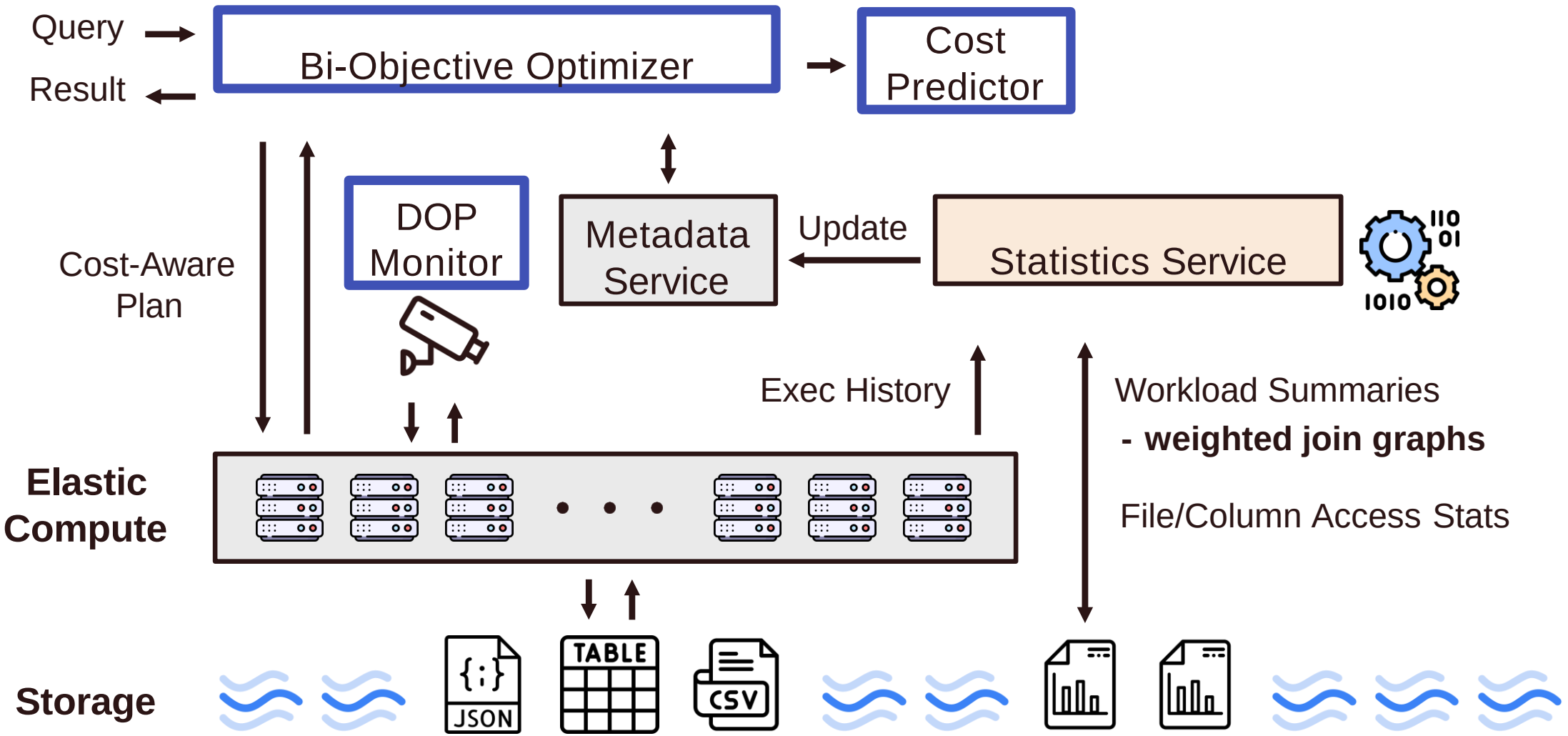
- ➜ Speeds up a subset of queries
- ➜ MV update slows down writes

Read Perf: +

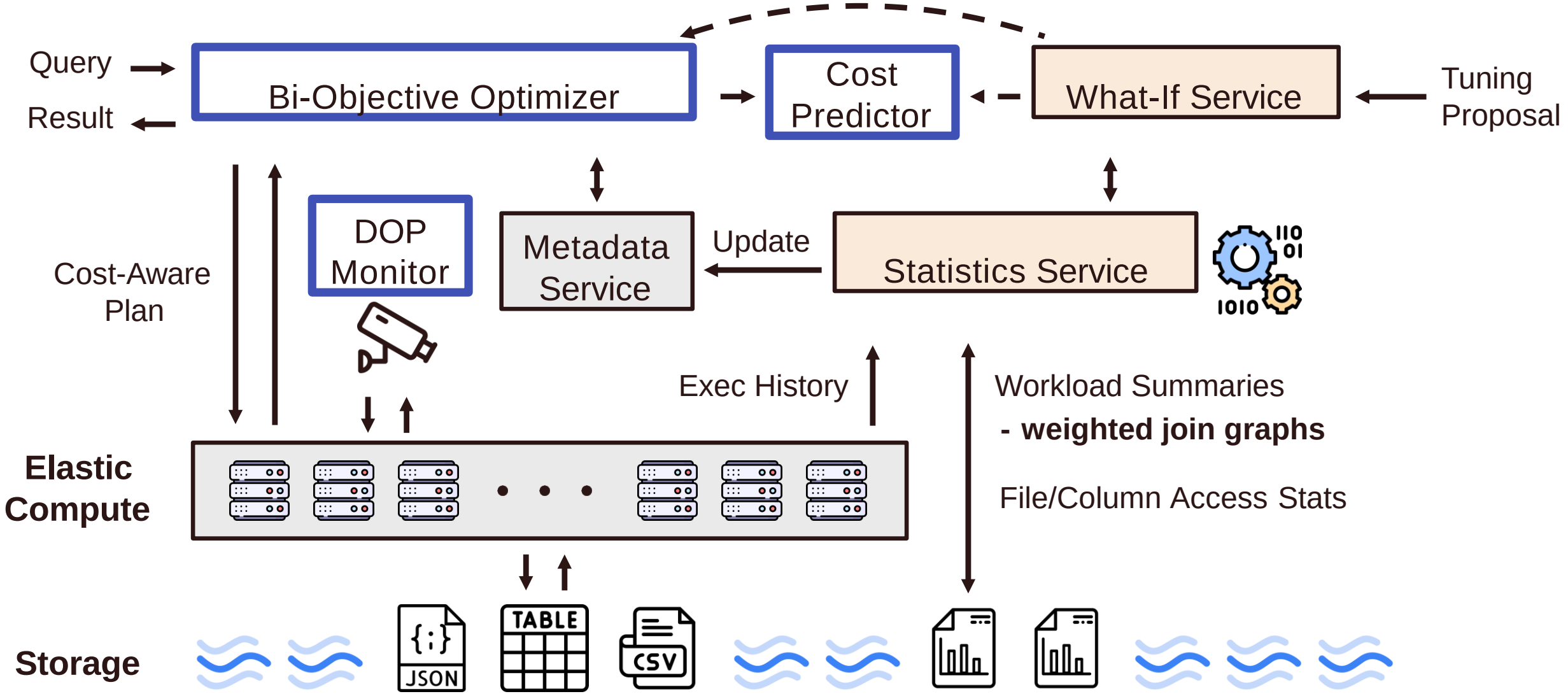
Write Perf: −

− **Resource Contention**

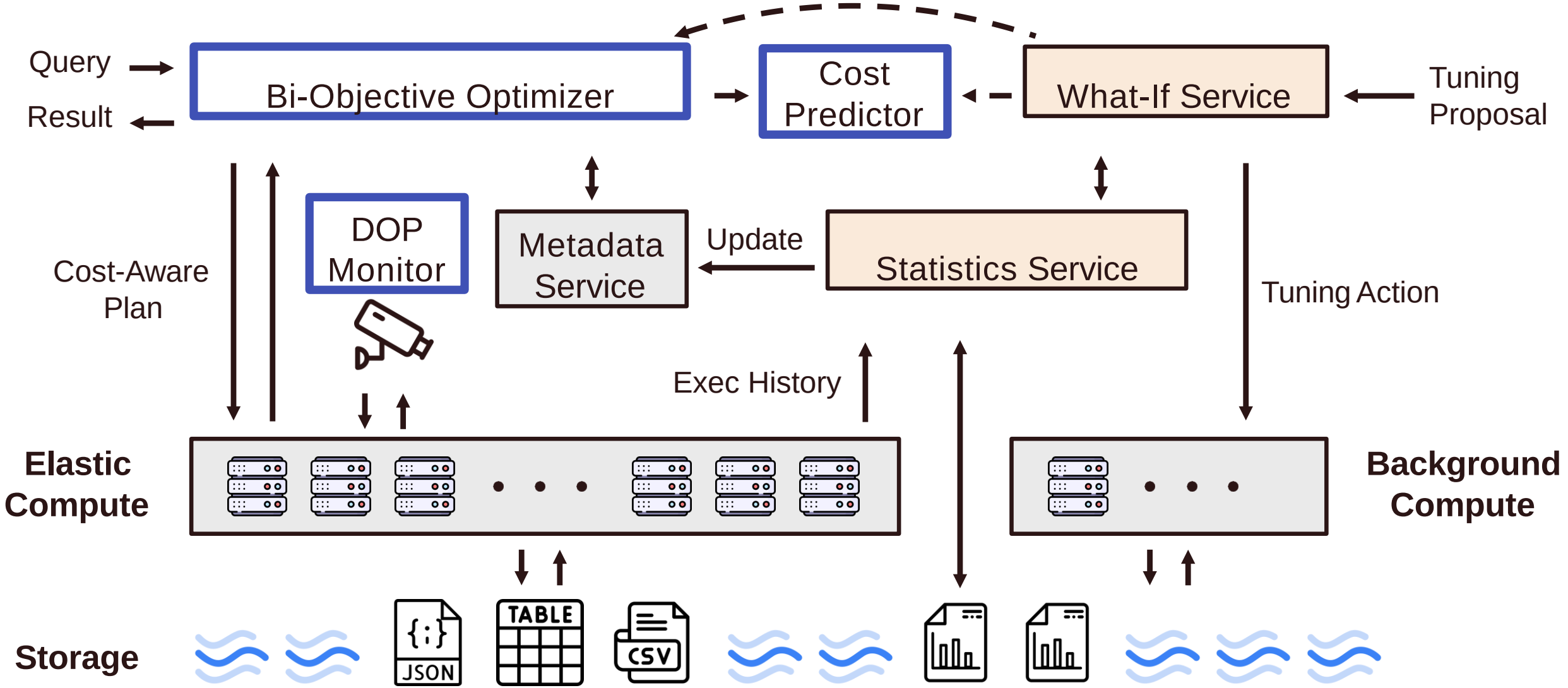
System architecture



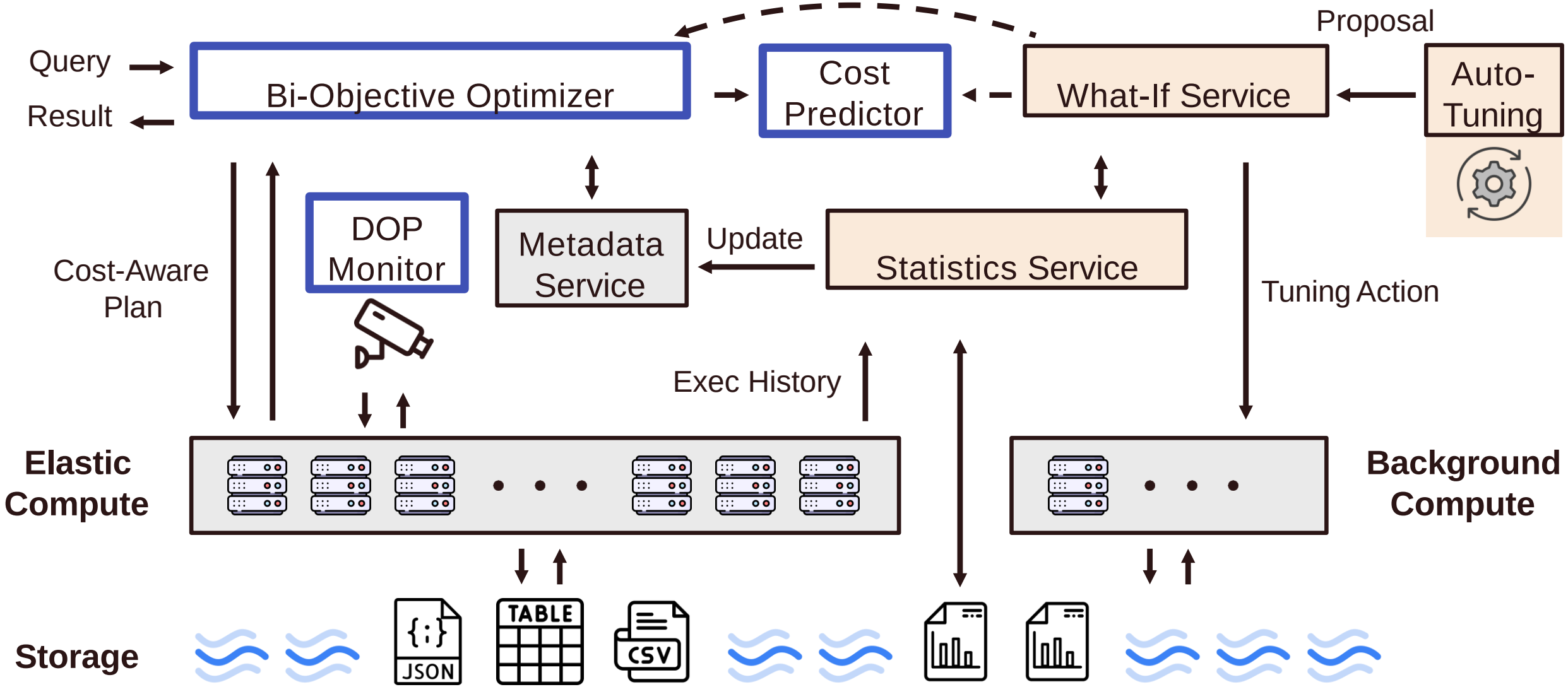
System architecture



System architecture

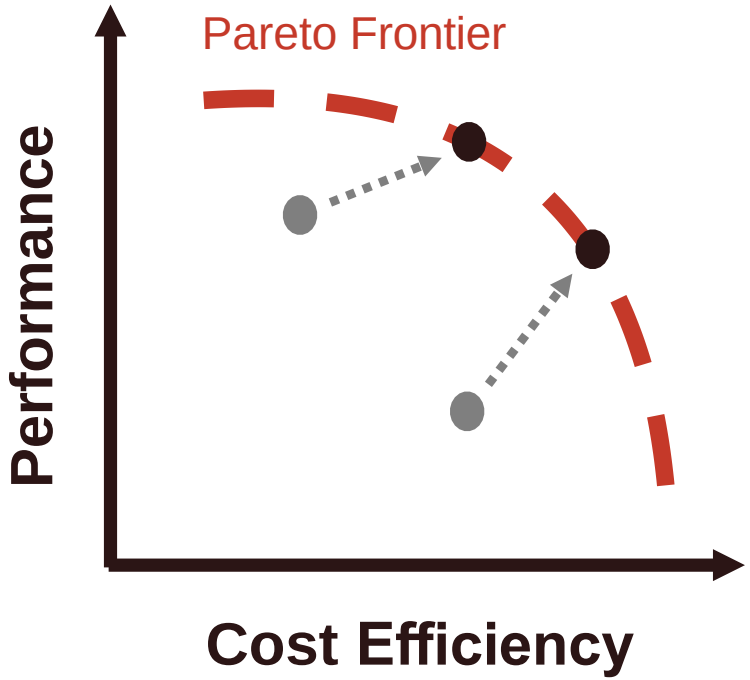


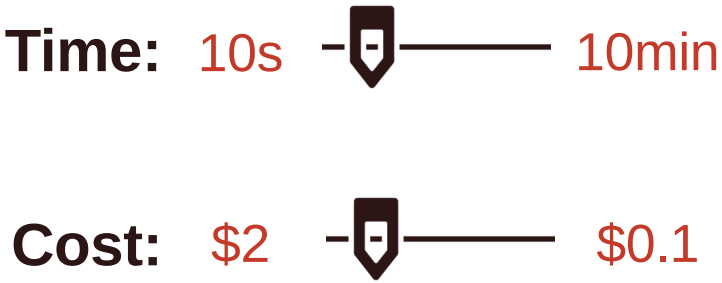
System architecture

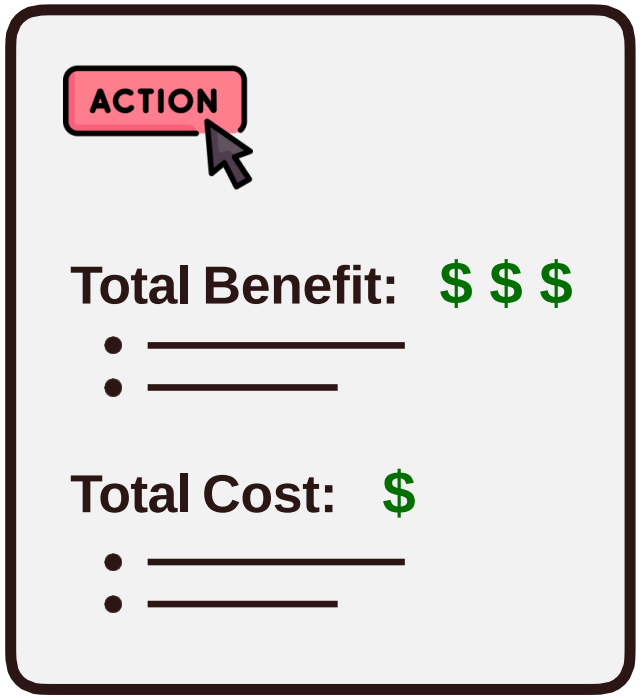


Towards cost intelligence

Cost is as important as performance in cloud-native databases

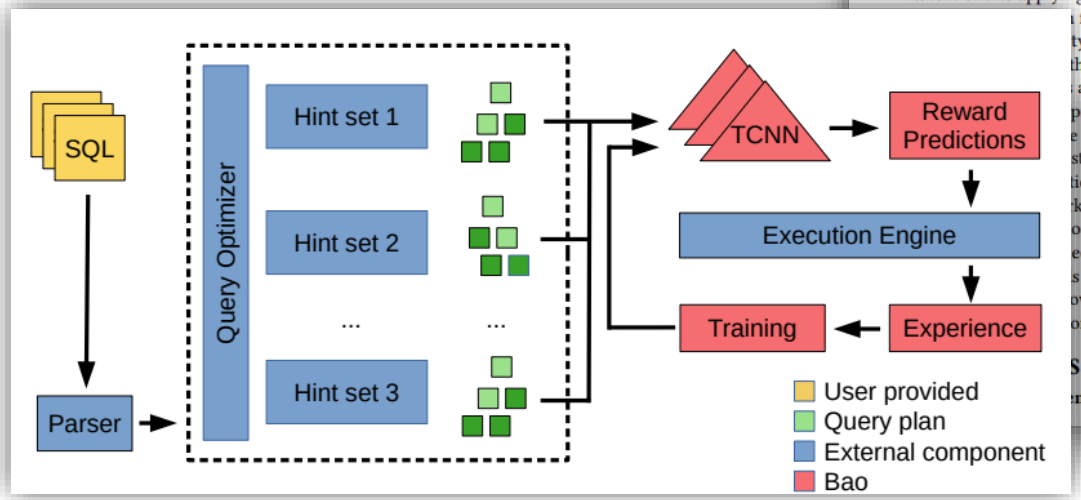


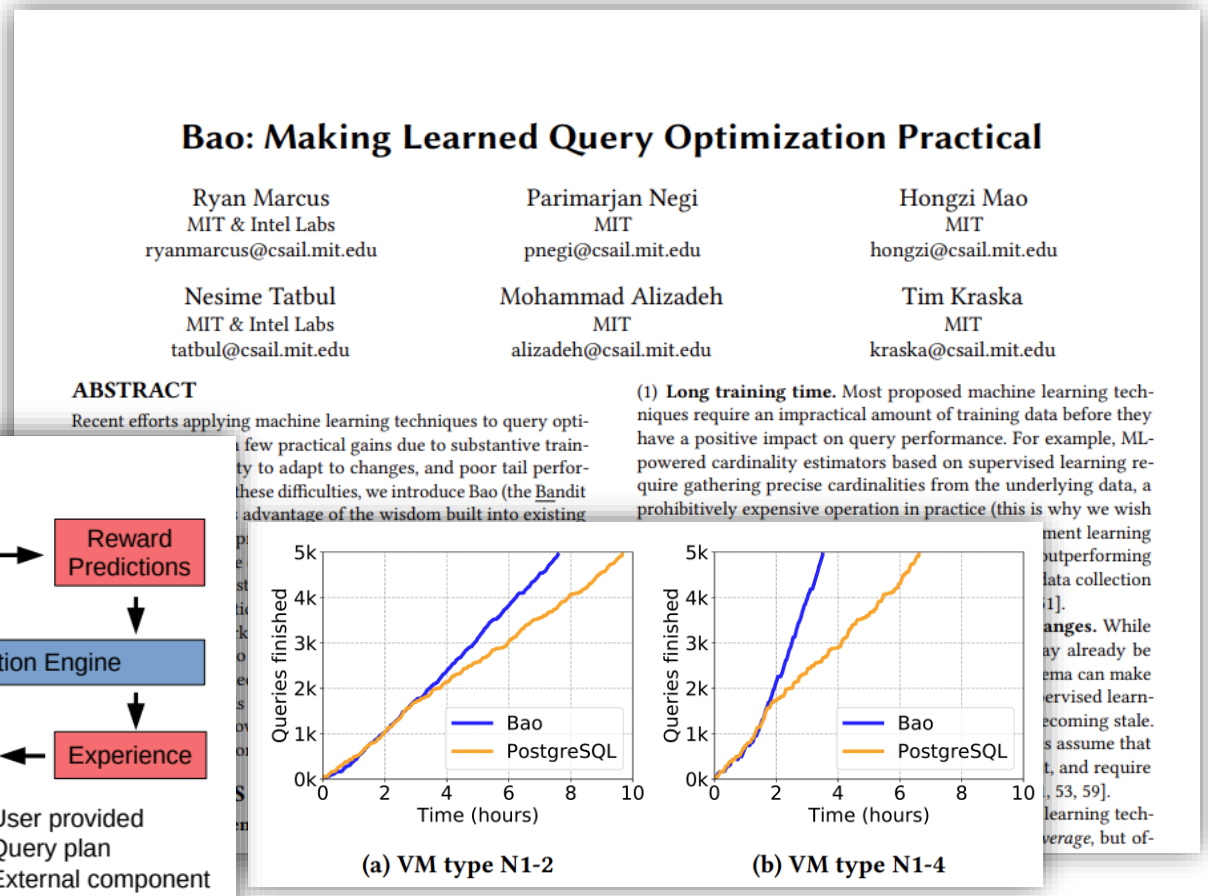




AI-enhanced system components

- QO hint recommendation on workload
  - Build a model for query hints
  - SIGMOD best paper award





Bao: Making Learned Query Optimization Practical

Ryan Marcus
MIT & Intel Labs
ryanmarcus@csail.mit.edu

Parimarjan Negi
MIT
pnegi@csail.mit.edu

Hongzi Mao
MIT
hongzi@csail.mit.edu

Nesime Tatbul
MIT & Intel Labs
tatbul@csail.mit.edu

Mohammad Alizadeh
MIT
alizadeh@csail.mit.edu

Tim Kraska
MIT
kraska@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query opti-
few practical gains due to substantive train-
ty to adapt to changes, and poor tail perfor-
these difficulties, we introduce Bao (the Bandit
advantage of the wisdom built into existing

(1) **Long training time.** Most proposed machine learning tech-niques require an impractical amount of training data before they have a positive impact on query performance. For example, ML-powered cardinality estimators based on supervised learning re-quire gathering precise cardinalities from the underlying data, a prohibitively expensive operation in practice (this is why we wish
ment learning
outperforming
lata collection
1].
anges. While
ay already be
ema can make
ervised learn-
ecoming stale.
s assume that
t, and require
., 53, 59].
learning tech-
verage, but of-

AI-enhanced system components

- Index recommendation on workload
- Workload compression
- Workload forecasting
  - Reducing what-if calls
  - Performance regression

ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges

Tarique Siddiqui Wentao Wu
Microsoft Research
{tasidd, wentwu}@microsoft.com

ABSTRACT

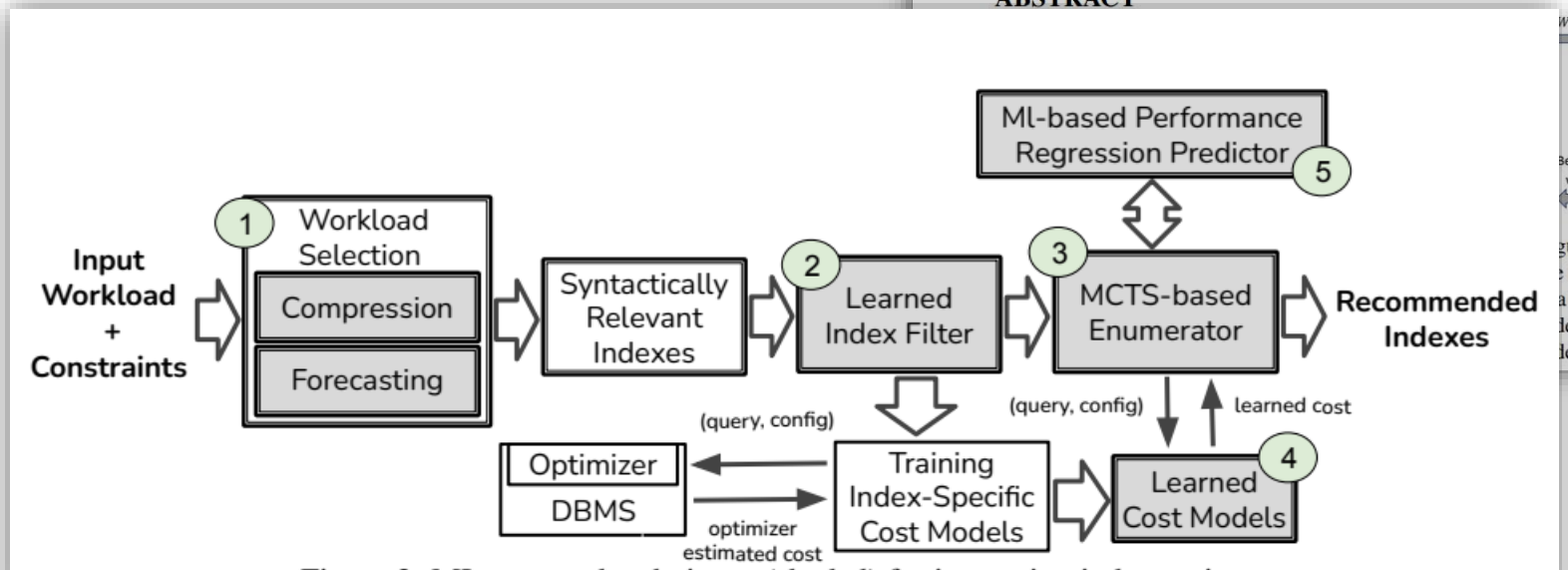
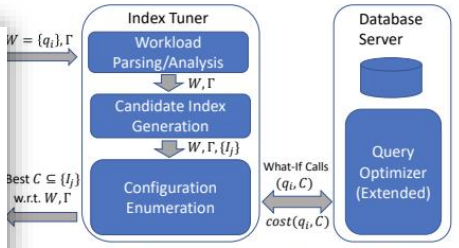


Figure 3: ML-powered techniques (shaded) for improving index tuning.



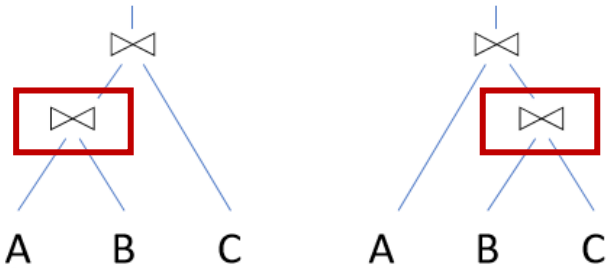
gure 1: The architecture of an index tuner, where W is input workload and $q_i \in W$ is a single SQL query, Γ a set of tuning constraints, $\{I_j\}$ is the set of candidate dexes generated for W, and $C \subseteq \{I_j\}$ represents an dex configuration during enumeration.

Query optimization & cardinality estimation

```
SELECT * FROM A,B,C
ON A.t = B.t = C.t
WHERE A.a=X and B.b=Y and C.c=Z
```

Two plans:

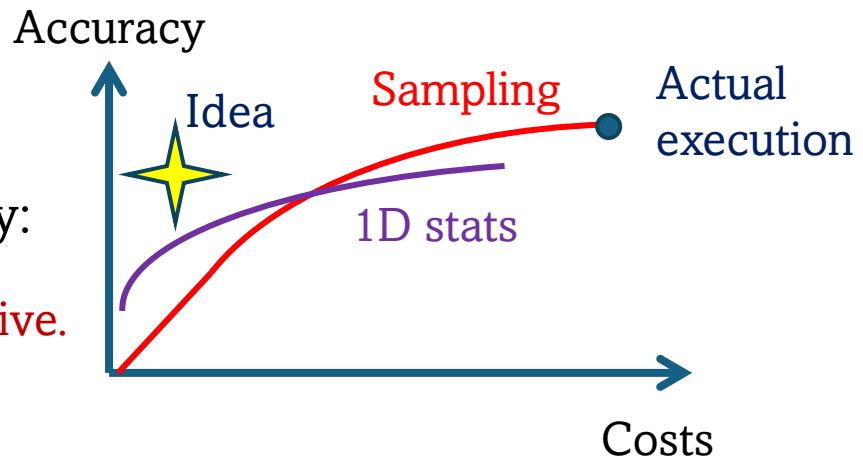
⋈ = join



Cost = 120 Cost = 17

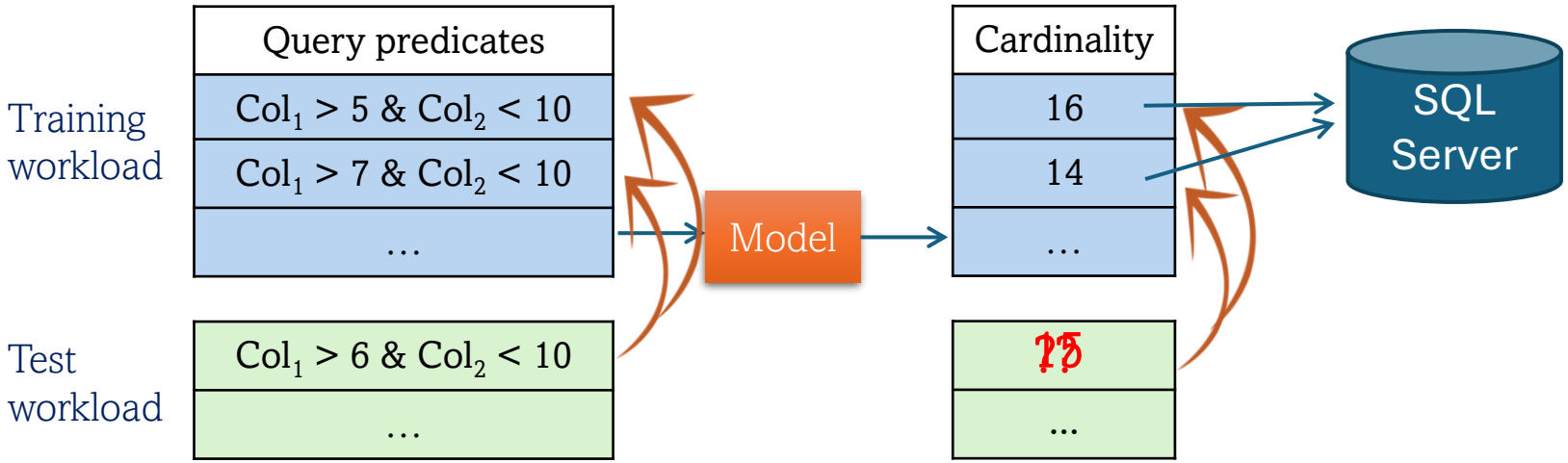
Cheaper if the first join has fewer output rows (cardinality).

- Fast & accurate cardinality estimation is essential:
  - A wrong plan may result in1000x performance regression.
- Cardinality estimation: a tradeoff between costs & accuracy:
  - Actual execution / multi-dim histograms / samples – expensive.
  - Production systems: column-wise stats – less accurate.



ML-based cardinality estimation

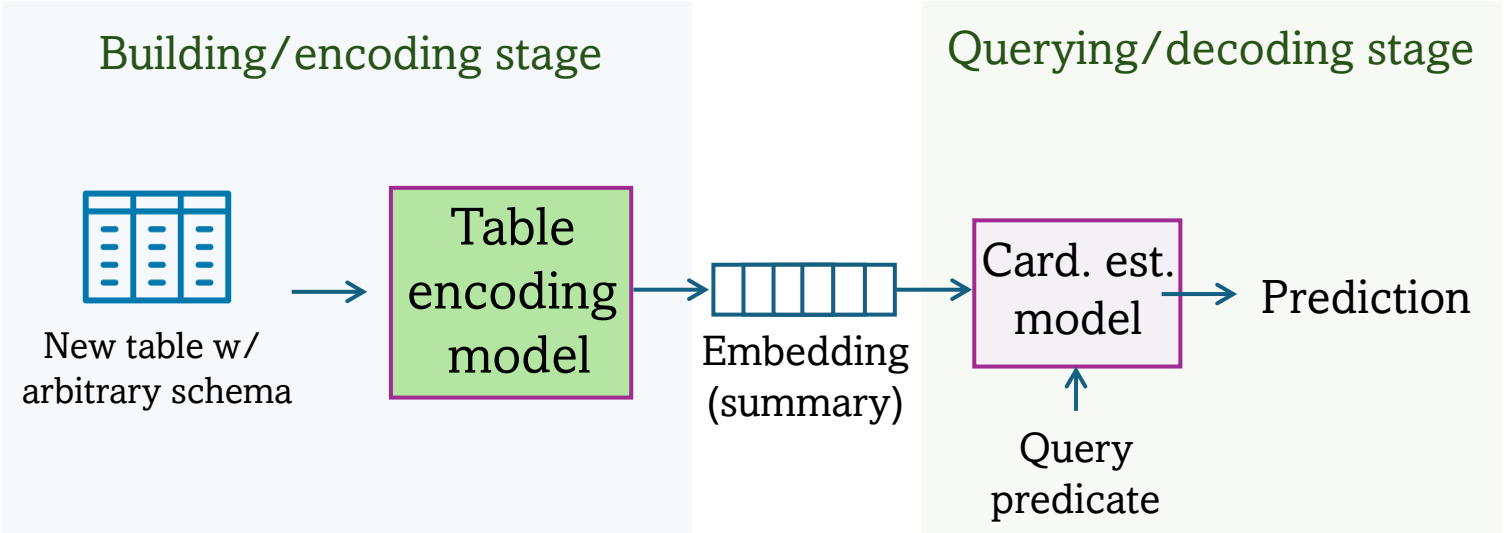
- Recent ML-enhanced cardinality estimation solutions are instance-optimized.
  - Train a model per workload: MSCN [CIDR 18], Learned Models [VLDB 19, 20].
  - Costly to label & to train models. Dataset/workload changes → retrain.



- Useful cardinality estimation:

  - Small building latency.
  - Adapting to changes quickly.
  - μs latency per cardinality estimate.
  - High accuracy.

Pretrained models structured tables



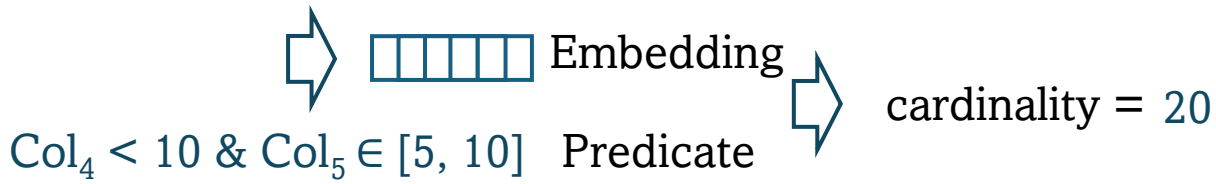
Use the models off-the-shelf as a part of the software for cardinality estimation.

- ✓ Small building latency.
- ▪ Adapting to changes quickly.
- ✓ μs latency per cardinality estimate.
- • High accuracy.

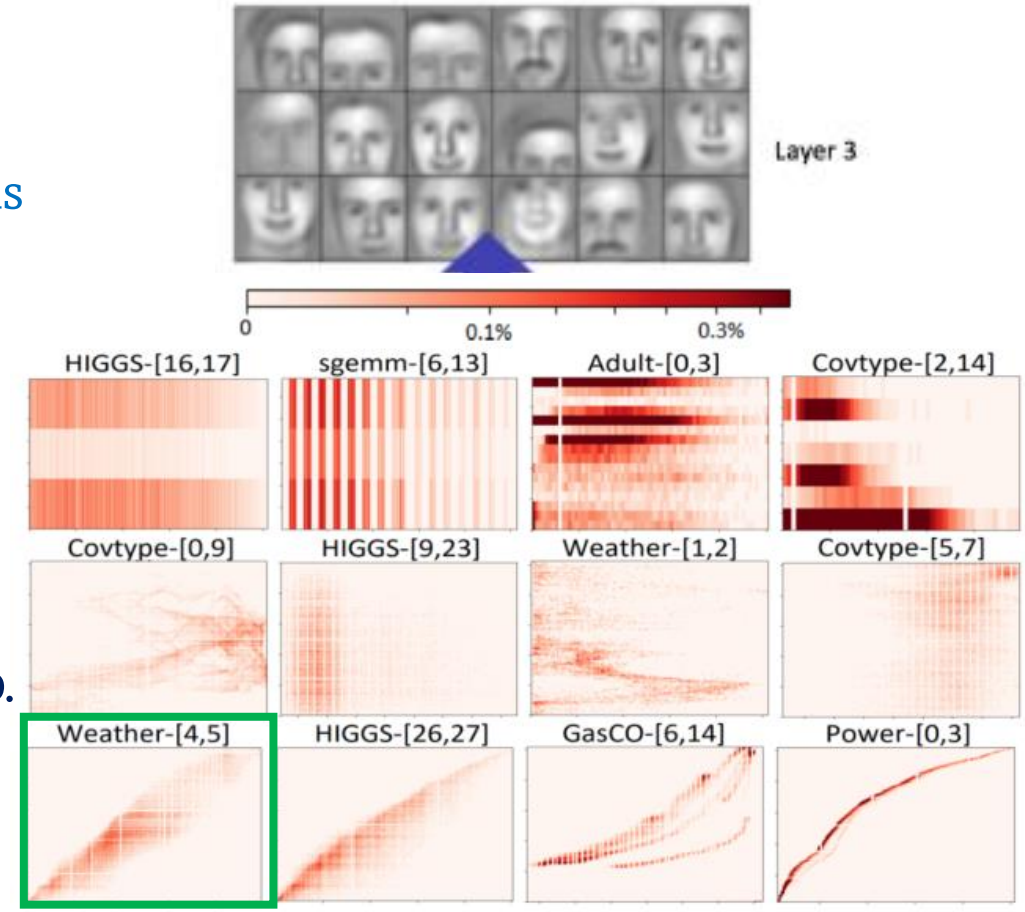
Yao Lu, Srikanth Kandula, Arnd Christian Konig, Surajit Chaudhuri. Pre-training Summarization Models of Structured Datasets for Cardinality Estimation. VLDB 2022.

Frequency distribution & correlation patterns in tables

- Images and text have common patterns.
- Accurate cardinality estimation depends on:
  - Multi-dim frequency distribution & correlation patterns (not string semantics).
- Example: [Weather Col_4, Weather Col_5]



- Pretrained on 10Ks of relations from UCI ML repo.
  - Similar/better accuracy @ small storage budget.
  - 1-2 orders of magnitude faster to build. Per estimate in μs.
  - Better cardinality estimation = end-to-end performance gains.



Yao Lu, Srikanth Kandula, Arnd Christian Konig, Surajit Chaudhuri. Pre-training Summarization Models of Structured Datasets for Cardinality Estimation. VLDB 2022.

Today’s agenda

- Current practice in distributed OLAP
- Cost-intelligent data analytics in the cloud

Credits

- Huanchen Zhang, Tsinghua
- Andy Pavlo, CMU