

01

Course Intro & Relational Model



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ORACLE®



TODAY'S AGENDA

Wait List

Overview

Course Logistics

Relational Model

Relational Algebra



WAIT LIST

There are currently 150 people on the waiting list.
Max capacity is 100.

We will enroll people based on your S3 position.



COURSE OVERVIEW

This course is on the design and implementation of disk-oriented database management systems.

This is not a course on how to use a database to build applications or how to administer a database.
→ See [CMU 95-703](#) (Heinz College)

Database Applications ([15-415/615](#)) is not offered this semester.

COURSE OUTLINE

Relational Databases

Storage

Execution

Concurrency Control

Recovery

Distributed Databases

Potpourri



COURSE LOGISTICS

Course Policies + Schedule:

→ Refer to [course web page](#).

Academic Honesty:

→ Refer to [CMU policy page](#).

→ If you're not sure, ask the professors.

→ Don't be stupid.

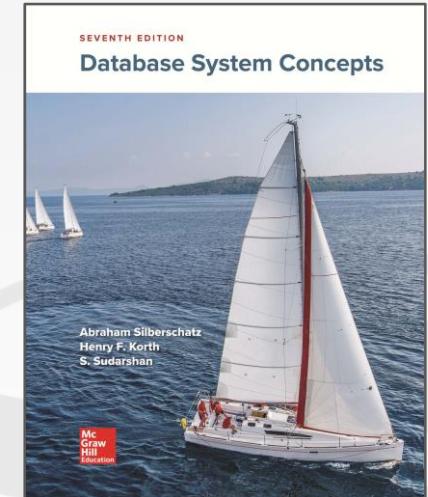
All discussion + announcements will be on [Piazza](#).



TEXTBOOK

Database System Concepts
7th Edition
Silberschatz, Korth, & Sudarshan

We will also provide lecture notes
that covers topics not found in textbook.



COURSE RUBRIC

Homeworks (15%)

Projects (45%)

Midterm Exam (20%)

Final Exam (20%)

Extra Credit (+10%)

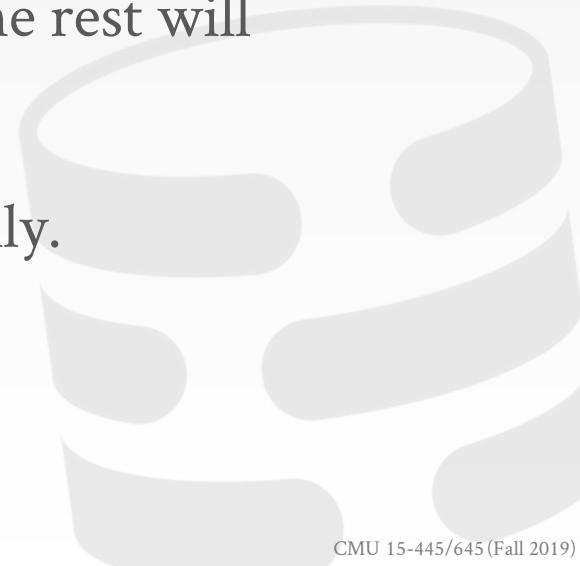


HOMEWORKS

Five homework assignments throughout the semester.

First homework is a SQL assignment. The rest will be pencil-and-paper assignments.

All homework should be done individually.

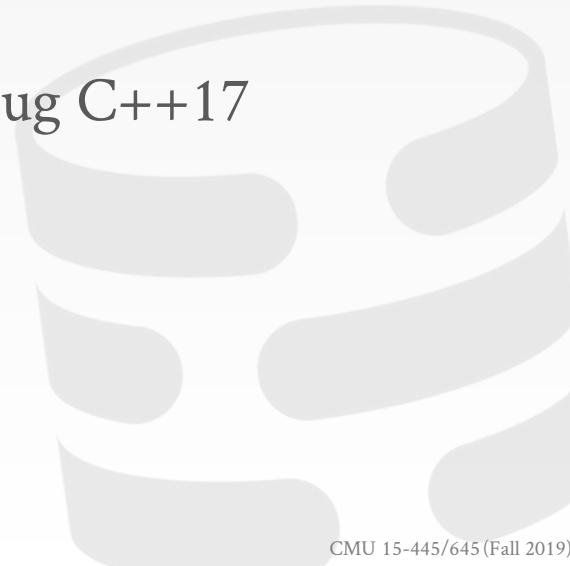


PROJECTS

You will build your own storage manager from scratch of the course of the semester.

Each project builds on the previous one.

We will not teach you how to write/debug C++17



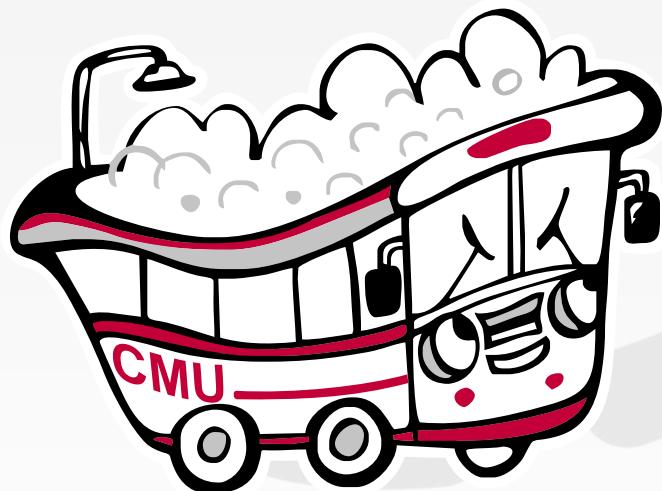
BUSTUB

All projects will use the new **BusTub** academic DBMS.

→ Source code will be released on Github.

Architecture:

- Disk-Oriented Storage
- Volcano-style Query Processing
- Pluggable APIs
- Currently does not support SQL.



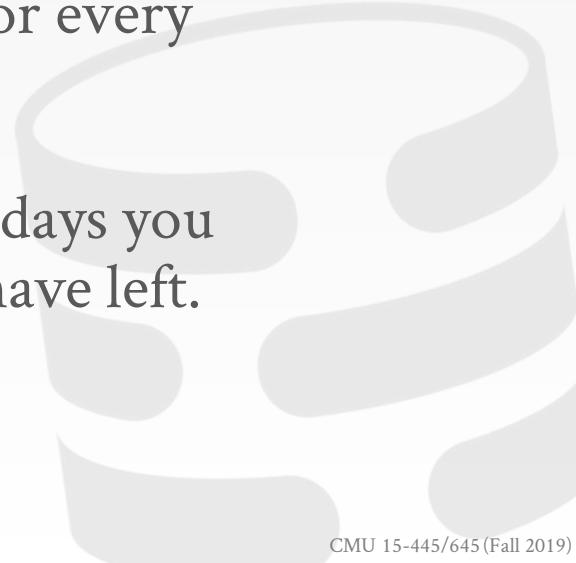
BusTub

LATE POLICY

You are allowed **four** slip days for either homework or projects.

You lose 25% of an assignment's points for every 24hrs it is late.

Mark on your submission (1) how many days you are late and (2) how many late days you have left.





PLAGIARISM WARNING



The homework and projects must be your own work. They are **not** group assignments.

You may **not** copy source code from other people or the web.

Plagiarism will **not** be tolerated.

See [CMU's Policy on Academic Integrity](#) for additional information.

DATABASE RESEARCH

Database Group Meetings

- Mondays @ 4:30pm (GHC 8102)
- <https://db.cs.cmu.edu>

Advanced DBMS Developer Meetings

- Tuesdays @ 12:00pm (GHC 8115)
- <https://github.com/cmu-db/terrier>



Databases

DATABASE

Organized collection of inter-related data that models some aspect of the real-world.

Databases are core the component of most computer applications.



DATABASE EXAMPLE

Create a database that models a digital music store to keep track of artists and albums.

Things we need store:

- Information about Artists
- What Albums those Artists released



FLAT FILE STRAWMAN

Store our database as comma-separated value (CSV) files that we manage in our own code.

- Use a separate file per entity.
- The application has to parse the files each time they want to read/update records.



FLAT FILE STRAWMAN

Create a database that models a digital music store.

Artist(name, year, country)

"Wu Tang Clan", 1992, "USA"

"Notorious BIG", 1992, "USA"

"Ice Cube", 1989, "USA"

Album(name, artist, year)

"Enter the Wu Tang", "Wu Tang Clan", 1993

"St. Ides Mix Tape", "Wu Tang Clan", 1994

"AmeriKKKa's Most Wanted", "Ice Cube", 1990

FLAT FILE STRAWMAN

Example: Get the year that Ice Cube went solo.

Artist(name, year, country)

"Wu Tang Clan", 1992, "USA"

"Notorious BIG", 1992, "USA"

"Ice Cube", 1989, "USA"



```
for line in file:  
    record = parse(line)  
    if "Ice Cube" == record[0]:  
        print int(record[1])
```

FLAT FILES: DATA INTEGRITY

How do we ensure that the artist is the same for each album entry?

What if somebody overwrites the album year with an invalid string?

How do we store that there are multiple artists on an album?

FLAT FILES: IMPLEMENTATION

How do you find a particular record?

What if we now want to create a new application
that uses the same database?

What if two threads try to write to the same file at
the same time?

FLAT FILES: DURABILITY

What if the machine crashes while our program is updating a record?

What if we want to replicate the database on multiple machines for high availability?



DATABASE MANAGEMENT SYSTEM

A **DBMS** is software that allows applications to store and analyze information in a database.

A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases.

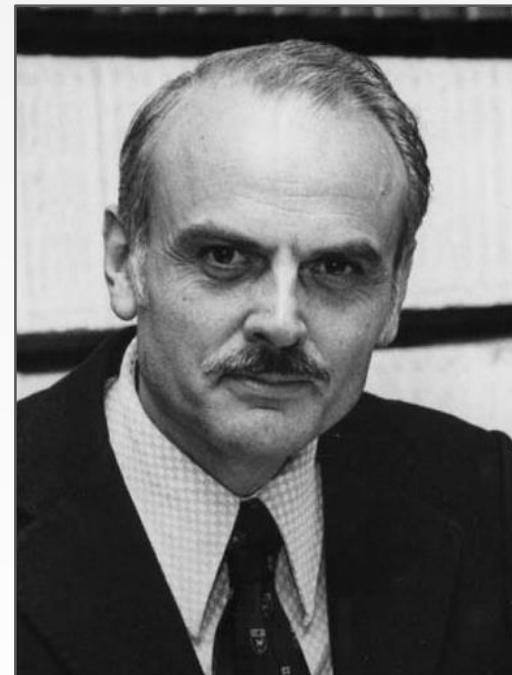


EARLY DBMSs

Database applications were difficult to build and maintain.

Tight coupling between logical and physical layers.

You have to (roughly) know what queries your app would execute before you deployed the database.



Edgar F. Codd

DERIVABILITY, REDUNDANCY AND CONSISTENCY OF RELATIONS
STORED IN LARGE DATA BANKS

E. F. Codd
Research Division
San Jose, California

ABSTRACT: The large, integrated data banks of the future will contain many relations of various degrees in stored form. It will not be unusual for this set of stored relations to be redundant. Two types of redundancy are defined and discussed. One type may be employed to improve accessibility of certain kinds of information which happen to be in great demand. When either type of redundancy exists, those responsible for control of the data bank should know about it and have some means of detecting any "logical" inconsistencies in the total set of stored relations. Consistency checking might be helpful in tracking down unauthorized (and possibly fraudulent) changes in the data bank contents.

RJ 599 (# 12343) August 19, 1969

LIMITED DISTRIBUTION NOTICE - This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from IBM Thomas J. Watson Research Center, Post Office Box 2118, Yorktown Heights, New York 10598

Information Retrieval

P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity.

CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

1. Relational Model and Normal Form

1.1. INTRODUCTION

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Leven and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for noninferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. DATA DEPENDENCIES IN PRESENT SYSTEMS

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed without logically impairing some application programs is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

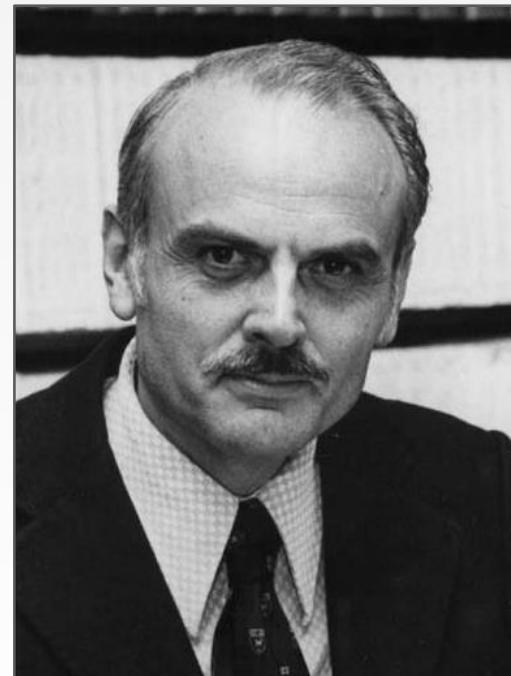
1.2.1. *Ordering Dependence.* Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

RELATIONAL MODEL

Proposed in 1970 by Ted Codd.

Database abstraction to avoid this maintenance:

- Store database in simple data structures.
- Access data through high-level language.
- Physical storage left up to implementation.



Edgar F. Codd

DATA MODELS

A data model is collection of concepts for describing the data in a database.

A schema is a description of a particular collection of data, using a given data model.



DATA MODEL

Relational

← Most DBMSs

Key/Value

Graph

Document

Column-family

Array / Matrix

Hierarchical

Network



DATA MODEL

Relational

Key/Value

Graph

Document

Column-family

Array / Matrix

Hierarchical

Network

← NoSQL



DATA MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network

← Machine Learning



DATA MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network

← Obsolete / Rare



DATA MODEL

Relational

← This Course

Key/Value

Graph

Document

Column-family

Array / Matrix

Hierarchical

Network

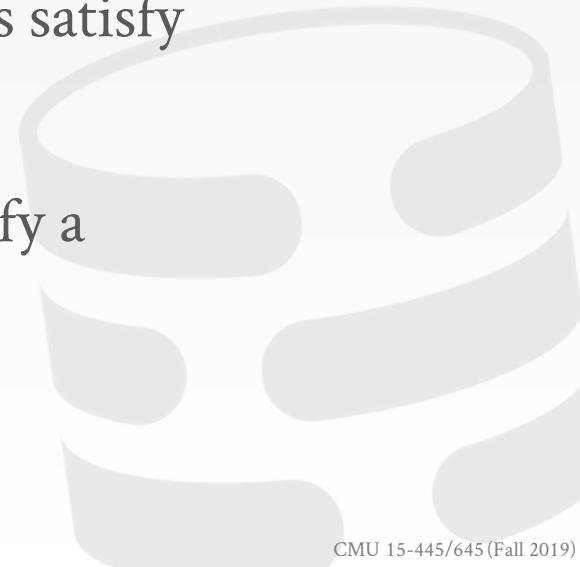


RELATIONAL MODEL

Structure: The definition of relations and their contents.

Integrity: Ensure the database's contents satisfy constraints.

Manipulation: How to access and modify a database's contents.



RELATIONAL MODEL

A relation is unordered set that contain the relationship of attributes that represent entities.

A tuple is a set of attribute values (also known as its domain) in the relation.
 → Values are (normally) atomic/scalar.
 → The special value **NULL** is a member of every domain.

Artist(name, year, country)

name	year	country
Wu Tang Clan	1992	USA
Notorious BIG	1992	USA
Ice Cube	1989	USA

n-ary Relation
 =
 Table with n columns

RELATIONAL MODEL: PRIMARY KEYS

A relation's primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if you don't define one.

Auto-generation of unique integer primary keys:

- **SEQUENCE** (SQL:2003)
- **AUTO_INCREMENT** (MySQL)

Artist(name, year, country)

name	year	country
Wu Tang Clan	1992	USA
Notorious BIG	1992	USA
Ice Cube	1989	USA

RELATIONAL MODEL: PRIMARY KEYS

A relation's primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if you don't define one.

Auto-generation of unique integer primary keys:

- **SEQUENCE** (SQL:2003)
- **AUTO_INCREMENT** (MySQL)

Artist(id, name, year, country)

id	name	year	country
123	Wu Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	Ice Cube	1989	USA

RELATIONAL MODEL: FOREIGN KEYS

A foreign key specifies that an attribute from one relation has to map to a tuple in another relation.



RELATIONAL MODEL: FOREIGN KEYS

Artist(id, name, year, country)

id	name	year	country
123	Wu Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	Ice Cube	1989	USA

Album(id, name, artists, year)

id	name	artists	year
11	Enter the Wu Tang	123	1993
22	St. Ides Mix Tape	???	1994
33	AmeriKKKa's Most Wanted	789	1990

RELATIONAL MODEL: FOREIGN KEYS

Artist(id, name, year, country)

id	name	year	country
123	Wu Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	Ice Cube	1989	USA

ArtistAlbum(artist_id, album_id)

artist_id	album_id
123	11
123	22
789	22
456	22

Album(id, name, artists, year)

id	name	artists	year
11	Enter the Wu Tang	123	1993
22	St. Ides Mix Tape	???	1994
33	AmeriKKKa's Most Wanted	789	1990

RELATIONAL MODEL: FOREIGN KEYS

Artist(id, name, year, country)

id	name	year	country
123	Wu Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	Ice Cube	1989	USA

ArtistAlbum(artist_id, album_id)

artist_id	album_id
123	11
123	22
789	22
456	22

Album(id, name, year)

id	name	year
11	<u>Enter the Wu Tang</u>	1993
22	<u>St. Ides Mix Tape</u>	1994
33	<u>AmeriKKKa's Most Wanted</u>	1990

RELATIONAL MODEL: FOREIGN KEYS

ArtistAlbum(artist_id, album_id)

artist_id	album_id
123	11
123	22
789	22
456	22

Artist(id, name, year, country)

id	name	year	country
123	Wu Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	Ice Cube	1989	USA

Album(id, name, year)

id	name	year
11	<u>Enter the Wu Tang</u>	1993
22	<u>St. Ides Mix Tape</u>	1994
33	<u>AmeriKKKa's Most Wanted</u>	1990

DATA MANIPULATION LANGUAGES (DML)

How to store and retrieve information from a database.

Procedural:

- The query specifies the (high-level) strategy the DBMS should use to find the desired result.

← Relational Algebra

Non-Procedural:

- The query specifies only what data is wanted and not how to find it.

DATA MANIPULATION LANGUAGES (DML)

How to store and retrieve information from a database.

Procedural:

- The query specifies the (high-level) strategy the DBMS should use to find the desired result.

← Relational Algebra

Non-Procedural:

- The query specifies only what data is wanted and not how to find it.

← Relational Calculus

RELATIONAL ALGEBRA

Fundamental operations to retrieve and manipulate tuples in a relation.
→ Based on set algebra.

Each operator takes one or more relations as its inputs and outputs a new relation.
→ We can “chain” operators together to create more complex operations.

σ	Select
π	Projection
\cup	Union
\cap	Intersection
$-$	Difference
\times	Product
\bowtie	Join

RELATIONAL ALGEBRA: SELECT

Choose a subset of the tuples from a relation that satisfies a selection predicate.

- Predicate acts as a filter to retain only tuples that fulfill its qualifying requirement.
- Can combine multiple predicates using conjunctions / disjunctions.

Syntax: $\sigma_{\text{predicate}}(R)$

$\sigma_{\text{a_id}='a2'}(R)$

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\sigma_{\text{a_id}='a2' \wedge \text{b_id}>102}(R)$

a_id	b_id
a2	102
a2	103

$\sigma_{\text{a_id}='a2' \wedge \text{b_id}>102}(R)$

a_id	b_id
a2	103

```
SELECT * FROM R
WHERE a_id='a2' AND b_id>102;
```

RELATIONAL ALGEBRA: PROJECTION

Generate a relation with tuples that contains only the specified attributes.

- Can rearrange attributes' ordering.
- Can manipulate the values.

Syntax: $\Pi_{A_1, A_2, \dots, A_n}(R)$

$R(a_id, b_id)$

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\Pi_{b_id=100, a_id}(\sigma_{a_id='a2'}(R))$

b_id=100	a_id
2	a2
3	a2

```
SELECT b_id=100, a_id
FROM R WHERE a_id = 'a2';
```

RELATIONAL ALGEBRA: UNION

Generate a relation that contains all tuples that appear in either only one or both input relations.

Syntax: **(R \cup S)**

```
(SELECT * FROM R)
UNION ALL
(SELECT * FROM S);
```

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R \cup S)

a_id	b_id
a1	101
a2	102
a3	103
a3	103
a4	104
a5	105

RELATIONAL ALGEBRA: INTERSECTION

Generate a relation that contains only the tuples that appear in both of the input relations.

Syntax: $(R \cap S)$

$R(a_id, b_id)$

a_id	b_id
a1	101
a2	102
a3	103

$S(a_id, b_id)$

a_id	b_id
a3	103
a4	104
a5	105

$(R \cap S)$

a_id	b_id
a3	103

```
(SELECT * FROM R)
INTERSECT
(SELECT * FROM S);
```

RELATIONAL ALGEBRA: DIFFERENCE

Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.

Syntax: $(R - S)$

$R(a_id, b_id)$

a_id	b_id
a1	101
a2	102
a3	103

$S(a_id, b_id)$

a_id	b_id
a3	103
a4	104
a5	105

$(R - S)$

a_id	b_id
a1	101
a2	102

```
(SELECT * FROM R)
EXCEPT
(SELECT * FROM S);
```

RELATIONAL ALGEBRA: PRODUCT

Generate a relation that contains all possible combinations of tuples from the input relations.

Syntax: $(R \times S)$

```
SELECT * FROM R CROSS JOIN S;
```

```
SELECT * FROM R, S;
```

$R(a_id, b_id)$

a_id	b_id
a1	101
a2	102
a3	103

$S(a_id, b_id)$

a_id	b_id
a3	103
a4	104
a5	105

$(R \times S)$

R.a_id	R.b_id	S.a_id	S.b_id
a1	101	a3	103
a1	101	a4	104
a1	101	a5	105
a2	102	a3	103
a2	102	a4	104
a2	102	a5	105
a3	103	a3	103
a3	103	a4	104
a3	103	a5	105

RELATIONAL ALGEBRA: JOIN

Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a **common** value(s) for one or more attributes.

Syntax: $(R \bowtie S)$

$R(a_id, b_id)$

a_id	b_id
a1	101
a2	102
a3	103

$S(a_id, b_id)$

a_id	b_id
a3	103
a4	104
a5	105

$(R \bowtie S)$

a_id	b_id
a3	103

SELECT * FROM R NATURAL JOIN S;

RELATIONAL ALGEBRA: EXTRA OPERATORS

Rename (ρ)

Assignment ($R \leftarrow S$)

Duplicate Elimination (δ)

Aggregation (γ)

Sorting (τ)

Division ($R \div S$)



OBSERVATION

Relational algebra still defines the high-level steps of how to compute a query.

→ $\sigma_{b_id=102}(R \bowtie S)$ vs. $(R \bowtie (\sigma_{b_id=102}(S)))$

A better approach is to state the high-level answer that you want the DBMS to compute.

→ Retrieve the joined tuples from **R** and **S** where **b_id** equals 102.

RELATIONAL MODEL: QUERIES

The relational model is independent of any query language implementation.

SQL is the *de facto* standard.

```
for line in file:  
    record = parse(line)  
    if "Ice Cube" == record[0]:  
        print int(record[1])
```

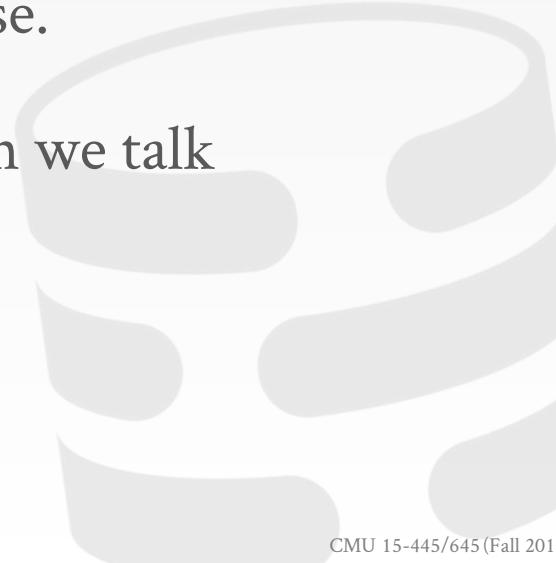
```
SELECT year FROM artists  
WHERE name = "Ice Cube";
```

CONCLUSION

Databases are ubiquitous.

Relational algebra defines the primitives for processing queries on a relational database.

We will see relational algebra again when we talk about query optimization + execution.



02

Advanced SQL



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

RELATIONAL LANGUAGES

User only needs to specify the answer that they want, not how to compute it.

The DBMS is responsible for efficient evaluation of the query.

→ Query optimizer: re-orders operations and generates query plan

SQL HISTORY

Originally “SEQUEL” from IBM’s
System R prototype.

- Structured English Query Language
- Adopted by Oracle in the 1970s.

IBM releases DB2 in 1983.

ANSI Standard in 1986. ISO in 1987
→ Structured Query Language



SQL HISTORY

Current standard is **SQL:2016**

- **SQL:2016** → JSON, Polymorphic tables
- **SQL:2011** → Temporal DBs, Pipelined DML
- **SQL:2008** → **TRUNCATE**, Fancy sorting
- **SQL:2003** → XML, windows, sequences, auto-gen IDs.
- **SQL:1999** → Regex, triggers, OO

Most DBMSs at least support **SQL-92**

- System Comparison: <http://troels.arvin.dk/db/rdbms/>

RELATIONAL LANGUAGES

Data Manipulation Language (DML)

Data Definition Language (DDL)

Data Control Language (DCL)

Also includes:

- View definition
- Integrity & Referential Constraints
- Transactions

Important: SQL is based on **bags** (duplicates) not **sets** (no duplicates).



TODAY'S AGENDA

Aggregations + Group By

String / Date / Time Operations

Output Control + Redirection

Nested Queries

Common Table Expressions

Window Functions



EXAMPLE DATABASE

student(sid,name,login,gpa)

sid	name	login	age	gpa
53666	Kanye	kayne@cs	39	4.0
53688	Bieber	jbieber@cs	22	3.9
53655	Tupac	shakur@cs	26	3.5

course(cid,name)

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53655	15-445	B
53666	15-721	C

AGGREGATES

Functions that return a single value from a bag of tuples:

- **AVG(col)** → Return the average col value.
- **MIN(col)** → Return minimum col value.
- **MAX(col)** → Return maximum col value.
- **SUM(col)** → Return sum of values in col.
- **COUNT(col)** → Return # of values for col.



AGGREGATES

Aggregate functions can only be used in the **SELECT** output list.

Get # of students with a "@cs" login:

```
SELECT COUNT(login) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

AGGREGATES

Aggregate functions can only be used in the **SELECT** output list.

Get # of students with a "@cs" login:

```
SELECT COUNT(login) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

AGGREGATES

Aggregate functions can only be used in the **SELECT** output list.

Get # of students with a "@cs" login:

```
SELECT COUNT(login) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

```
SELECT COUNT(*) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

AGGREGATES

Aggregate functions can only be used in the **SELECT** output list.

Get # of students with a "@cs" login:

```
SELECT COUNT(login) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

```
SELECT COUNT(*) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

```
SELECT COUNT(1) AS cnt  
FROM student WHERE login LIKE '%@cs'
```

MULTIPLE AGGREGATES

Get the number of students and their average GPA that have a “@cs” login.

```
SELECT AVG(gpa), COUNT(sid)  
FROM student WHERE login LIKE '%@cs'
```

AVG(gpa)	COUNT(sid)
3.25	12

DISTINCT AGGREGATES

COUNT, SUM, AVG support DISTINCT

Get the number of unique students that have an "@cs" login.

```
SELECT COUNT(DISTINCT login)
FROM student WHERE login LIKE '%@cs'
```

COUNT(DISTINCT login)
10

AGGREGATES

Output of other columns outside of an aggregate is undefined.

Get the average GPA of students enrolled in each course.

```
SELECT AVG(s.gpa), e.cid  
FROM enrolled AS e, student AS s  
WHERE e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.5	???

GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid  
FROM enrolled AS e, student AS s  
WHERE e.sid = s.sid  
GROUP BY e.cid
```

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	15-721
53439	53439	2.70	15-721
56023	56023	2.75	15-826
59439	59439	3.90	15-826
53961	53961	3.50	15-826
58345	58345	1.89	15-445

GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
```

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	15-721
53439	53439	2.70	15-721
56023	56023	2.75	15-826
59439	59439	3.90	15-826
53961	53961	3.50	15-826
58345	58345	1.89	15-445



AVG(s.gpa)	e.cid
2.46	15-721
3.39	15-826
1.89	15-445

GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
```

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	15-721
53439	53439	2.70	15-721
56023	56023	2.75	15-826
59439	59439	3.90	15-826
53961	53961	3.50	15-826
58345	58345	1.89	15-445



AVG(s.gpa)	e.cid
2.46	15-721
3.39	15-826
1.89	15-445

GROUP BY

Non-aggregated values in **SELECT** output clause must appear in **GROUP BY** clause.

```
SELECT AVG(s.gpa), e.cid, s.name
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
```



GROUP BY

Non-aggregated values in **SELECT** output clause must appear in **GROUP BY** clause.

```
SELECT AVG(s.gpa), e.cid, s.name  
  FROM enrolled AS e, student AS s  
 WHERE e.sid = s.sid  
GROUP BY e.cid, s.name
```

HAVING

Filters results based on aggregation computation.

Like a **WHERE** clause for a **GROUP BY**

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid  
FROM enrolled AS e, student AS s  
WHERE e.sid = s.sid  
AND avg_gpa > 3.9  
GROUP BY e.cid
```



HAVING

Filters results based on aggregation computation.

Like a **WHERE** clause for a **GROUP BY**

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid  
  FROM enrolled AS e, student AS s  
 WHERE e.sid = s.sid  
 GROUP BY e.cid  
HAVING avg_gpa > 3.9;
```

HAVING

Filters results based on aggregation computation.

Like a **WHERE** clause for a **GROUP BY**

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid  
FROM enrolled AS e, student AS s  
WHERE e.sid = s.sid  
GROUP BY e.cid  
HAVING avg_gpa > 3.9;
```

AVG(s.gpa)	e.cid
3.75	15-415
3.950000	15-721
3.900000	15-826



avg_gpa	e.cid
3.950000	15-721

SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY

STRING OPERATIONS

	String Case	String Quotes
SQL-92	Sensitive	Single Only
Postgres	Sensitive	Single Only
MySQL	Insensitive	Single/Double
SQLite	Sensitive	Single/Double
DB2	Sensitive	Single Only
Oracle	Sensitive	Single Only

WHERE UPPER(name) = UPPER('KaNyE') **SQL-92**

WHERE name = "KaNyE" **MySQL**

STRING OPERATIONS

LIKE is used for string matching.

String-matching operators

- '%' Matches any substring (including empty strings).
- '_' Match any one character

```
SELECT * FROM enrolled AS e  
WHERE e.cid LIKE '15-%'
```

```
SELECT * FROM student AS s  
WHERE s.login LIKE '%@c_'
```

STRING OPERATIONS

SQL-92 defines string functions.

→ Many DBMSs also have their own unique functions

Can be used in either output and predicates:

```
SELECT SUBSTRING(name,0,5) AS abbrv_name  
FROM student WHERE sid = 53688
```

```
SELECT * FROM student AS s  
WHERE UPPER(e.name) LIKE 'KAN%'
```

STRING OPERATIONS

SQL standard says to use **||** operator to
concatenate two or more strings together.

```
SELECT name FROM student           SQL-92
      WHERE login = LOWER(name) || '@cs'
```

```
SELECT name FROM student           MSSQL
      WHERE login = LOWER(name) + '@cs'
```

```
SELECT name FROM student           MySQL
      WHERE login = CONCAT(LOWER(name), '@cs')
```

DATE/TIME OPERATIONS

Operations to manipulate and modify **DATE/TIME** attributes.

Can be used in either output and predicates.

Support/syntax varies wildly...

Demo: Get the # of days since the beginning of the year.

OUTPUT REDIRECTION

Store query results in another table:

- Table must not already be defined.
- Table will have the same # of columns with the same types as the input.

```
SELECT DISTINCT cid INTO CourseIds SQL-92
      FROM enrolled;
```

```
CREATE TABLE CourseIds (
      SELECT DISTINCT cid FROM enrolled); MySQL
```

OUTPUT REDIRECTION

Insert tuples from query into another table:

- Inner **SELECT** must generate the same columns as the target table.
- DBMSs have different options/syntax on what to do with duplicates.

```
INSERT INTO CourseIds          SQL-92  
  (SELECT DISTINCT cid FROM enrolled);
```

OUTPUT CONTROL

ORDER BY <column*> [ASC|DESC]

→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled  
WHERE cid = '15-721'  
ORDER BY grade
```

sid	grade
53123	A
53334	A
53650	B
53666	D

OUTPUT CONTROL

ORDER BY <column*> [ASC|DESC]

→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
WHERE cid = '15-721'
ORDER BY grade
```

sid	grade
53123	A
53334	A
53650	B
53666	D

```
SELECT sid FROM enrolled
WHERE cid = '15-721'
ORDER BY grade DESC, sid ASC
```

sid
53666
53650
53123
53334

OUTPUT CONTROL

LIMIT <count> [offset]

- Limit the # of tuples returned in output.
- Can set an offset to return a “range”

```
SELECT sid, name FROM student  
WHERE login LIKE '%@cs'  
LIMIT 10
```

OUTPUT CONTROL

LIMIT <count> [offset]

- Limit the # of tuples returned in output.
- Can set an offset to return a “range”

```
SELECT sid, name FROM student  
WHERE login LIKE '%@cs'  
LIMIT 10
```

```
SELECT sid, name FROM student  
WHERE login LIKE '%@cs'  
LIMIT 20 OFFSET 10
```

NESTED QUERIES

Queries containing other queries.

They are often difficult to optimize.

Inner queries can appear (almost) anywhere in query.

Outer Query →

```
SELECT name FROM student WHERE  
sid IN (SELECT sid FROM enrolled)
```

← Inner Query

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student  
WHERE ...
```



sid in the set of people that take 15-445

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student  
WHERE ...
```

```
    SELECT sid FROM enrolled  
    WHERE cid = '15-445'
```

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
WHERE sid IN (
    SELECT sid FROM enrolled
    WHERE cid = '15-445'
)
```

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student  
WHERE sid IN (  
    SELECT sid FROM enrolled  
    WHERE cid = '15-445'  
)
```

NESTED QUERIES

ALL→ Must satisfy expression for all rows in sub-query

ANY→ Must satisfy expression for at least one row in sub-query.

IN→ Equivalent to '**=ANY()**' .

EXISTS→ At least one row is returned.



NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
WHERE sid = ANY(
    SELECT sid FROM enrolled
    WHERE cid = '15-445'
)
```

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT (SELECT S.name FROM student AS S  
        WHERE S.sid = E.sid) AS sname  
      FROM enrolled AS E  
     WHERE cid = '15-445'
```

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT MAX(e.sid), s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid;
```

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT MAX(e.sid), s.name  
FROM enrolled AS e, student AS s  
WHERE e.sid = s.sid;
```



Won't work in SQL-92. This runs in SQLite, but not Postgres or MySQL (v5.7 with strict mode).

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT sid, name FROM student  
WHERE ...
```

"Is greater than every other sid"

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT sid, name FROM student  
WHERE sid is greater than every  
    SELECT sid FROM enrolled
```

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT sid, name FROM student
WHERE sid => ALL(
    SELECT sid FROM enrolled
)
```

sid	name
53688	Bieber

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT sid, name FROM student
WHERE sid = (
    SELECT sid
    FROM student
    WHERE sid IN (
        SELECT MAX(sid) FROM enrolled
    )
)
```

NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT sid, name FROM student
  WHERE sid = (
    SELECT sid, name FROM student
      WHERE sid = (
        SELECT sid, name FROM student
          WHERE sid IN (
            SELECT sid FROM enrolled
              ORDER BY sid DESC LIMIT 1
          )
      )
  )
```

NESTED QUERIES

Find all courses that has no students enrolled in it.

```
SELECT * FROM course  
WHERE ...
```

"with no tuples in the 'enrolled' table"

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53655	15-445	B
53666	15-721	C

NESTED QUERIES

Find all courses that has no students enrolled in it.

```
SELECT * FROM course  
WHERE NOT EXISTS(  
    tuples in the 'enrolled' table  
)
```

NESTED QUERIES

Find all courses that has no students enrolled in it.

```
SELECT * FROM course
WHERE NOT EXISTS(
    SELECT * FROM enrolled
    WHERE course.cid = enrolled.cid
)
```

cid	name
15-823	Advanced Topics in Databases

NESTED QUERIES

Find all courses that has no students enrolled in it.

```
SELECT * FROM course  
WHERE NOT EXISTS(  
    SELECT * FROM enrolled  
    WHERE course.cid = enrolled.cid  
)
```

cid	name
15-823	Advanced Topics in Databases

WINDOW FUNCTIONS

Performs a "sliding" calculation across a set of tuples that are related.

Like an aggregation but tuples are not grouped into a single output tuples.

```
SELECT ... FUNC-NAME(...) OVER (...)  
FROM tableName
```

WINDOW FUNCTIONS

Performs a "sliding" calculation across a set of tuples that are related.

Like an aggregation but tuples are not grouped into a single output tuples.

```
SELECT ... FUNC-NAME(...) OVER (...)  
FROM tableName
```

Aggregation Functions
Special Functions

How to "slice" up data
Can also sort

WINDOW FUNCTIONS

Aggregation functions:

→ Anything that we discussed earlier

Special window functions:

→ **ROW_NUMBER()** → # of the current row

→ **RANK()** → Order position of the current row.

sid	cid	grade	row_num
53666	15-445	C	1
53688	15-721	A	2
53688	15-826	B	3
53655	15-445	B	4
53666	15-721	C	5

```
SELECT *, ROW_NUMBER() OVER () AS row_num  
FROM enrolled
```

WINDOW FUNCTIONS

Aggregation functions:

→ Anything that we discussed earlier

Special window functions:

→ **ROW_NUMBER()** → # of the current row

→ **RANK()** → Order position of the current row.

sid	cid	grade	row_num
53666	15-445	C	1
53688	15-721	A	2
53688	15-826	B	3
53655	15-445	B	4
53666	15-721	C	5

```
SELECT *, ROW_NUMBER() OVER () AS row_num  
FROM enrolled
```

WINDOW FUNCTIONS

The **OVER** keyword specifies how to group together tuples when computing the window function.

Use **PARTITION BY** to specify group.

cid	sid	row_number
15-445	53666	1
15-445	53655	2
15-721	53688	1
15-721	53666	2
15-826	53688	1

```
SELECT cid, sid,  
       ROW_NUMBER() OVER (PARTITION BY cid)  
    FROM enrolled  
   ORDER BY cid
```

WINDOW FUNCTIONS

The **OVER** keyword specifies how to group together tuples when computing the window function.

Use **PARTITION BY** to specify group.

cid	sid	row_number
15-445	53666	1
15-445	53655	2
15-721	53688	1
15-721	53666	2
15-826	53688	1

```
SELECT cid, sid,  
       ROW_NUMBER() OVER (PARTITION BY cid)  
    FROM enrolled  
   ORDER BY cid
```

WINDOW FUNCTIONS

You can also include an **ORDER BY** in the window grouping to sort entries in each group.

```
SELECT *,  
    ROW_NUMBER() OVER (ORDER BY cid)  
FROM enrolled  
ORDER BY cid
```

WINDOW FUNCTIONS

Find the student with the highest grade for each course.

```
SELECT * FROM (
    SELECT *,  
        RANK() OVER (PARTITION BY cid  
                      ORDER BY grade ASC)  
    AS rank  
    FROM enrolled) AS ranking  
WHERE ranking.rank = 1
```

WINDOW FUNCTIONS

Find the student with the highest grade for each course.

```
SELECT * FROM (
    SELECT *,
        RANK() OVER (PARTITION BY cid
                      ORDER BY grade ASC)
        AS rank
    FROM enrolled) AS ranking
WHERE ranking.rank = 1
```

Group tuples by cid
Then sort by grade



WINDOW FUNCTIONS

Find the student with the highest grade for each course.

```
SELECT * FROM (
    SELECT *,  
        RANK() OVER (PARTITION BY cid  
                      ORDER BY grade ASC)  
    AS rank  
    FROM enrolled) AS ranking  
WHERE ranking.rank = 1
```

Group tuples by cid
Then sort by grade

COMMON TABLE EXPRESSIONS

Provides a way to write **auxiliary statements** for use in a larger query.

→ Think of it like a temp table just for one query.

Alternative to nested queries and views.

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName
```

COMMON TABLE EXPRESSIONS

Provides a way to write auxiliary statements for use in a larger query.

→ Think of it like a temp table just for one query.

Alternative to nested queries and views.

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName
```

COMMON TABLE EXPRESSIONS

You can bind output columns to names before the **AS** keyword.

```
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)
SELECT col1 + col2 FROM cteName
```

COMMON TABLE EXPRESSIONS

Find student record with the highest id that is enrolled in at least one course.

```
WITH cteSource (maxId) AS (
    SELECT MAX(sid) FROM enrolled
)
SELECT name FROM student, cteSource
WHERE student.sid = cteSource.maxId
```

COMMON TABLE EXPRESSIONS

Find student record with the highest id that is enrolled in at least one course.

```
WITH cteSource (maxId) AS (←  
    SELECT MAX(sid) FROM enrolled  
)  
SELECT name FROM student, cteSource  
WHERE student.sid = cteSource.maxId
```

CTE – RECURSION

Print the sequence of numbers from 1 to 10.

```
WITH RECURSIVE cteSource (counter) AS (
    (SELECT 1)
    UNION ALL
    (SELECT counter + 1 FROM cteSource
     WHERE counter < 10)
)
SELECT * FROM cteSource
```



Demo: Postgres CTE!

CONCLUSION

SQL is not a dead language.

You should (almost) always strive to compute your answer as a single SQL statement.



NEXT CLASS

Storage Management



03 |

Database Storage — Part I



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #1 is due September 11th @ 11:59pm

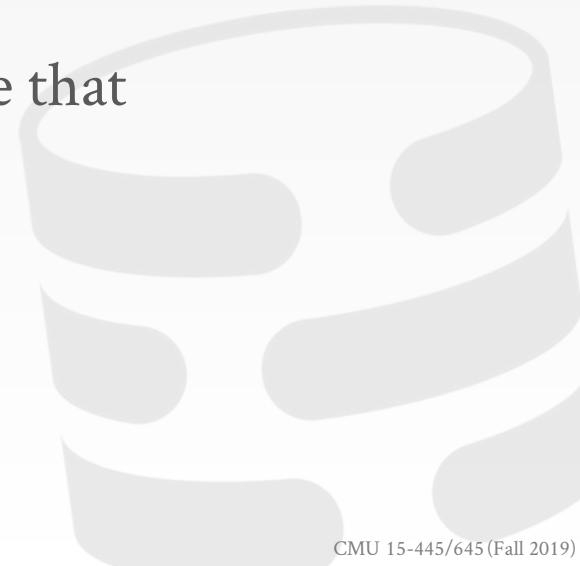
Project #1 will be released on September 11th



OVERVIEW

We now understand what a database looks like at a logical level and how to write queries to read/write data from it.

We will next learn how to build software that manages a database.



COURSE OUTLINE

Relational Databases
Storage
Execution
Concurrency Control
Recovery
Distributed Databases
Potpourri

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

DISK-ORIENTED ARCHITECTURE

The DBMS assumes that the primary storage location of the database is on non-volatile disk.

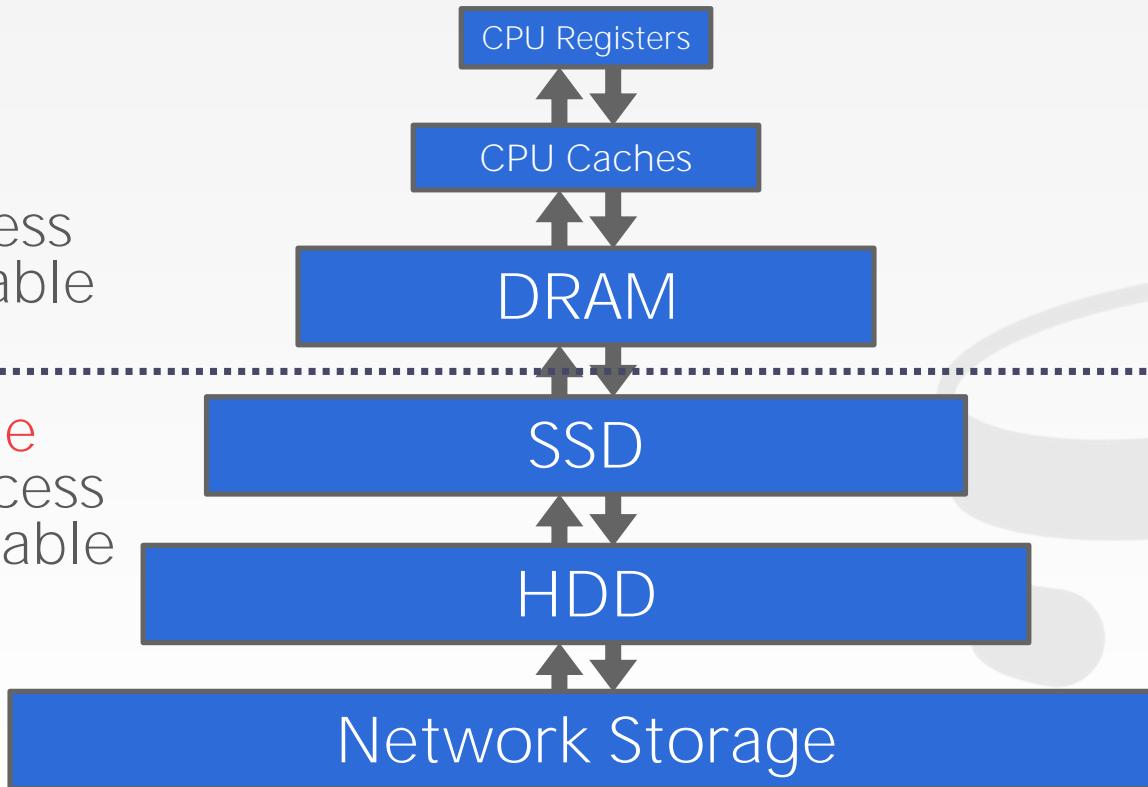
The DBMS's components manage the movement of data between non-volatile and volatile storage.



STORAGE HIERARCHY

Volatile
Random Access
Byte-Addressable

Non-Volatile
Sequential Access
Block-Addressable

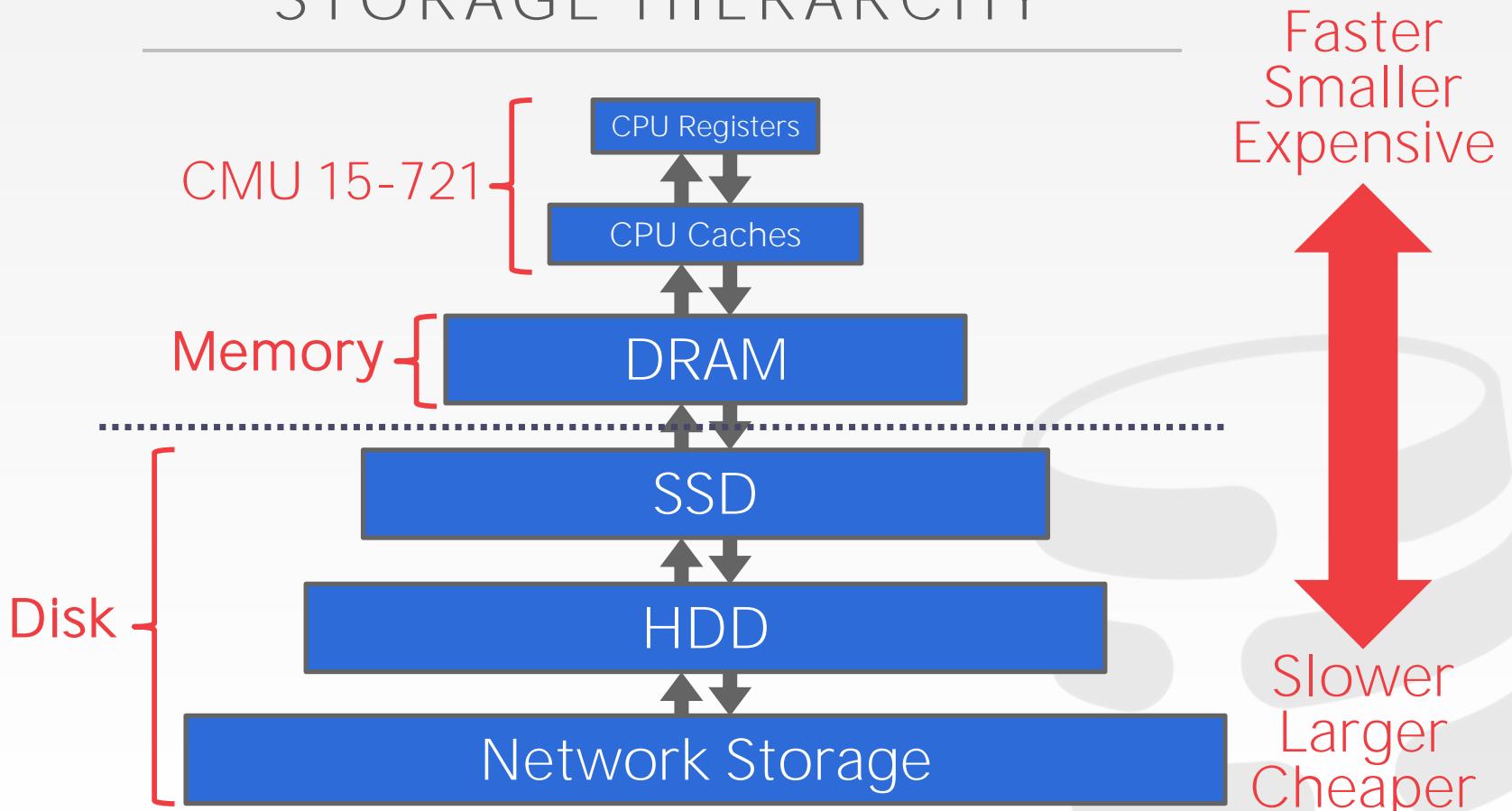


Faster
Smaller
Expensive

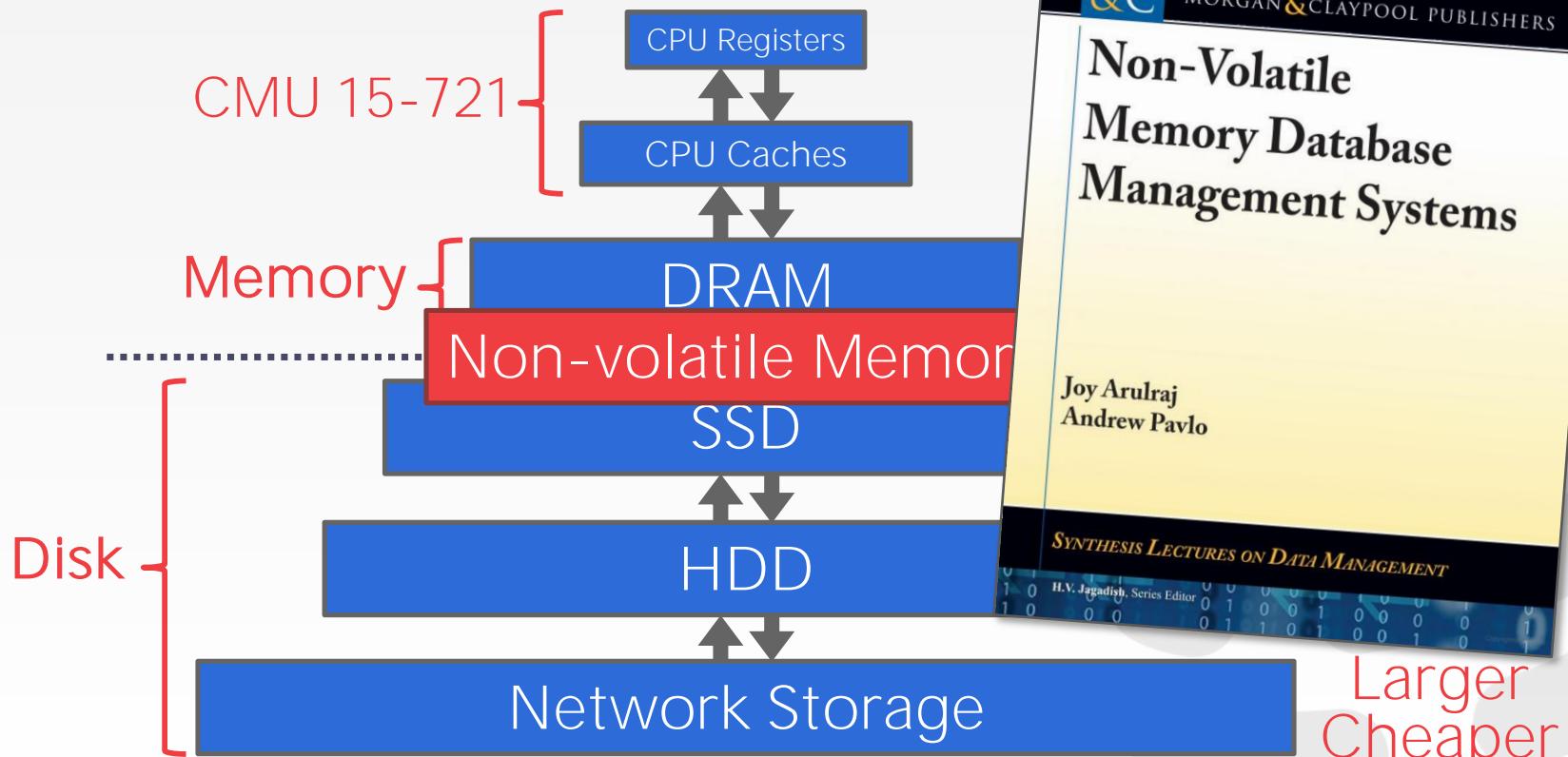


Slower
Larger
Cheaper

STORAGE HIERARCHY



STORAGE HIERARCHY



ACCESS TIMES

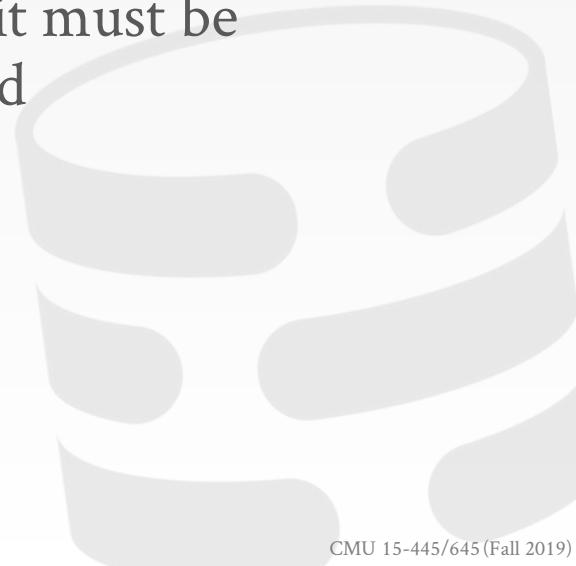
0.5 ns L1 Cache Ref	0.5 sec
7 ns L2 Cache Ref	7 sec
100 ns DRAM	100 sec
150,000 ns SSD	1.7 days
10,000,000 ns HDD	16.5 weeks
~30,000,000 ns Network Storage	11.4 months
1,000,000,000 ns Tape Archives	31.7 years

[Source]

SYSTEM DESIGN GOALS

Allow the DBMS to manage databases that exceed the amount of memory available.

Reading/writing to disk is expensive, so it must be managed carefully to avoid large stalls and performance degradation.

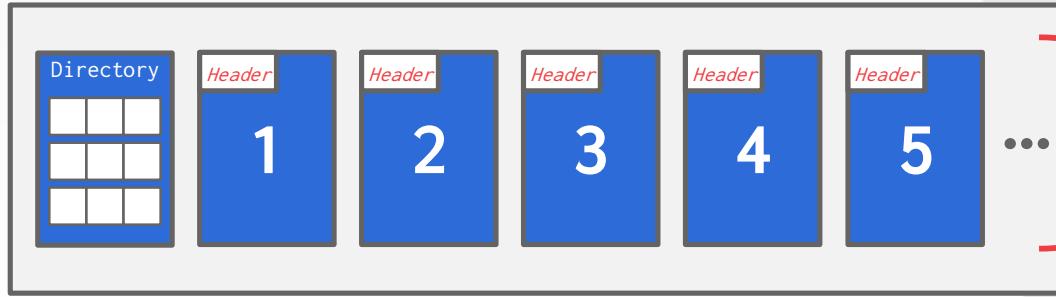


DISK-ORIENTED DBMS



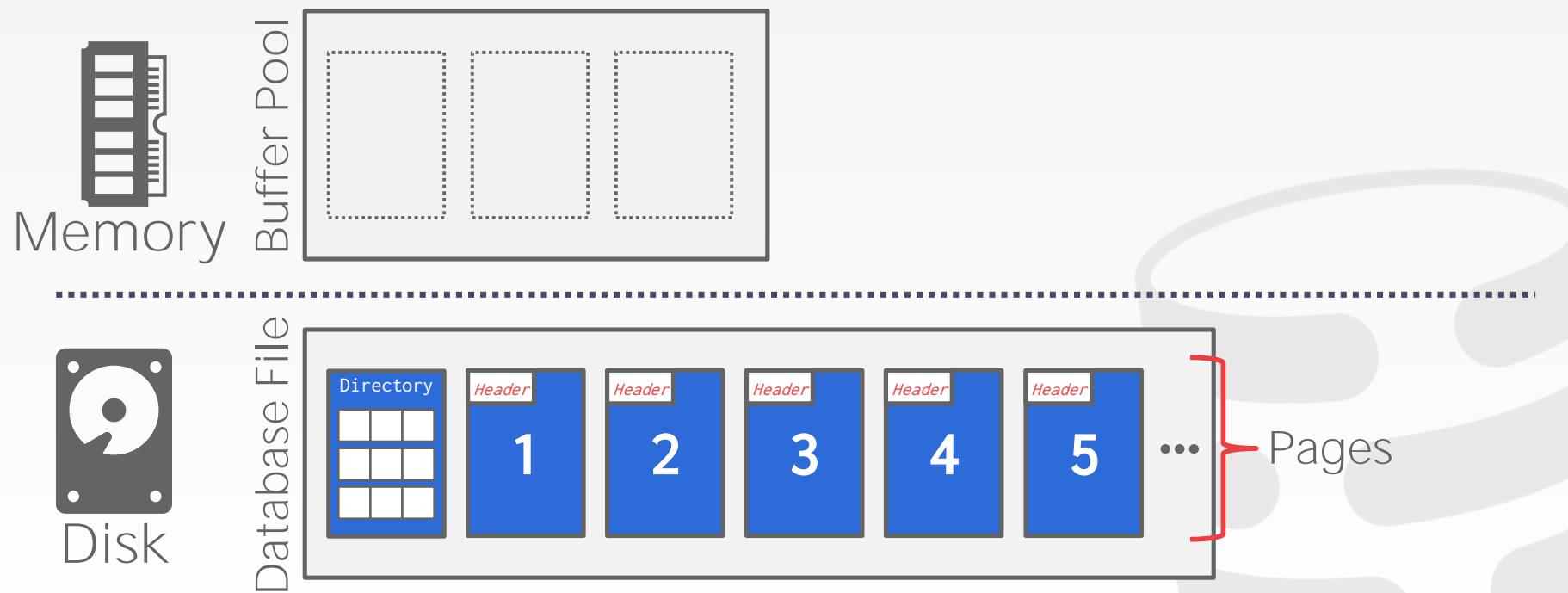
Disk

Database File

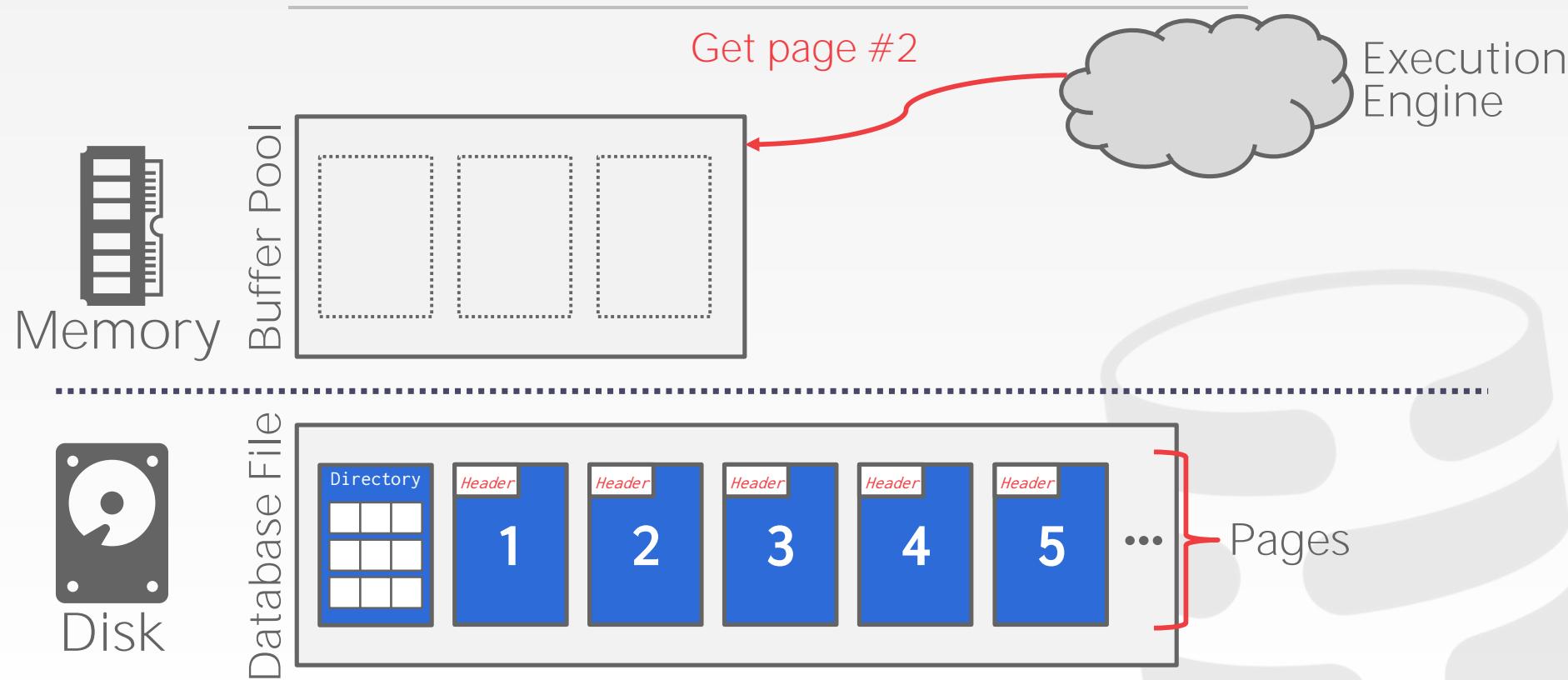


Pages

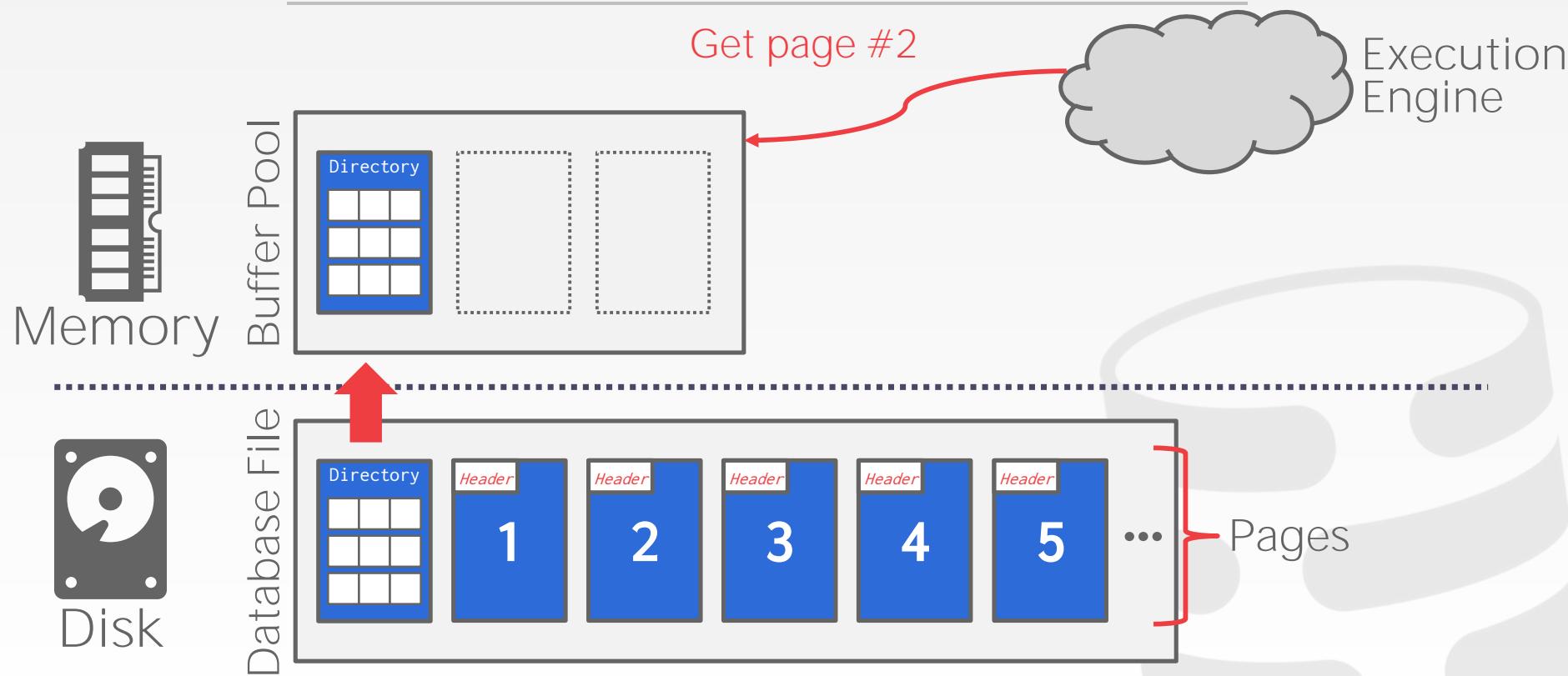
DISK-ORIENTED DBMS



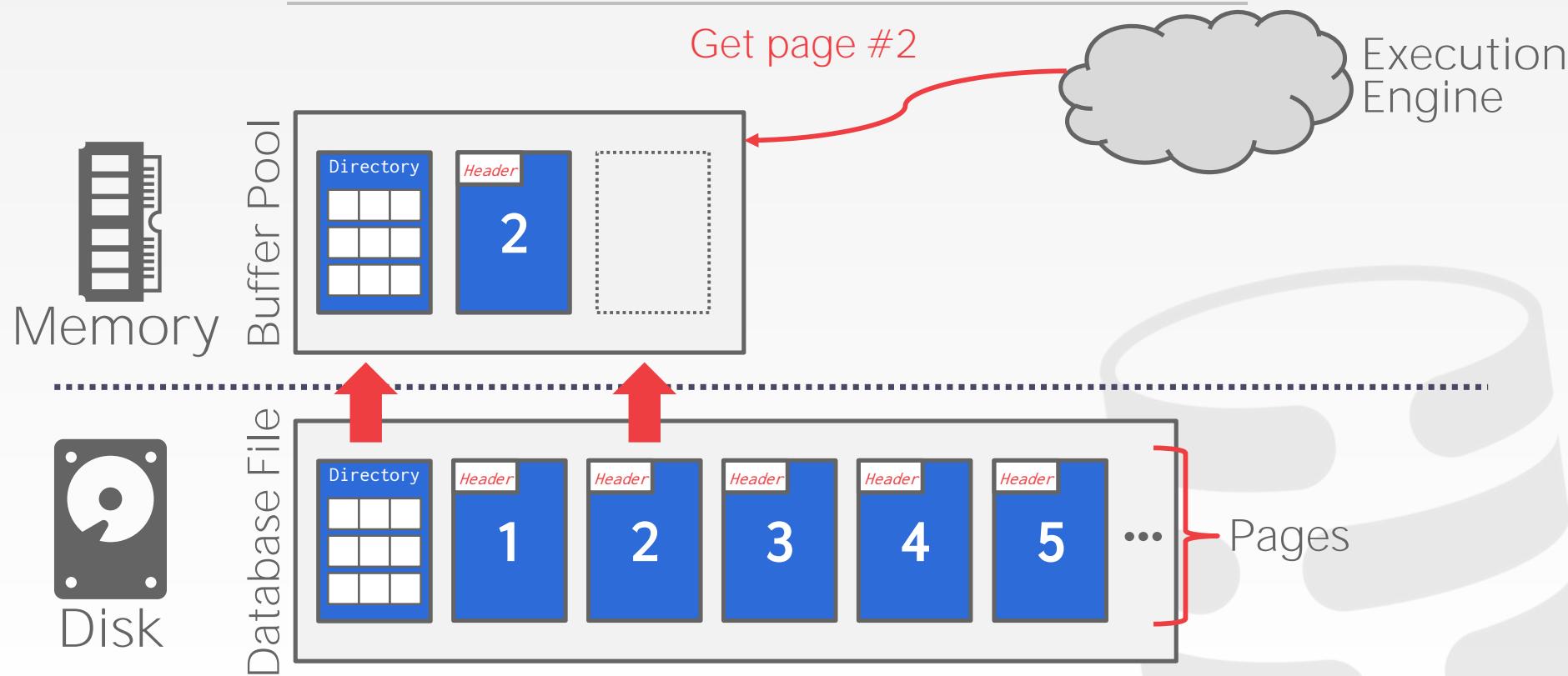
DISK-ORIENTED DBMS



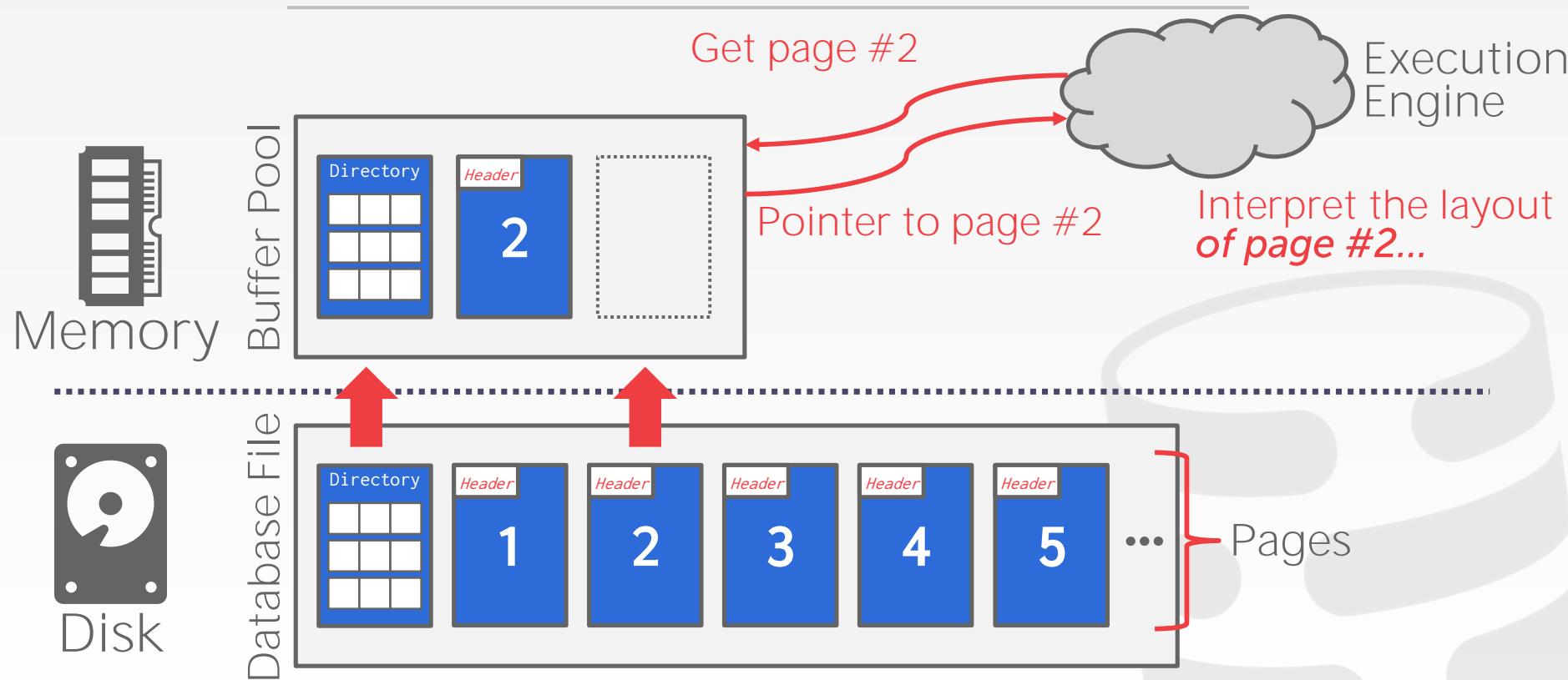
DISK-ORIENTED DBMS



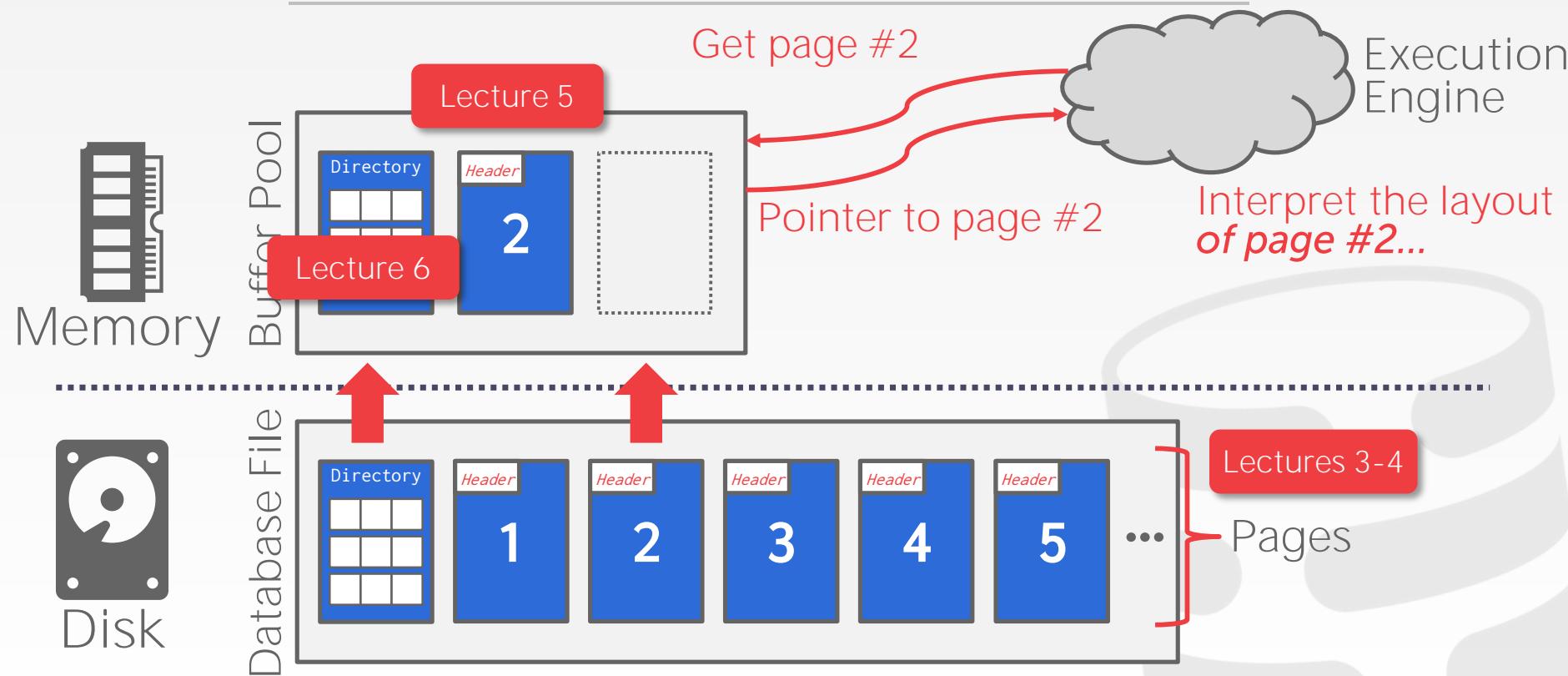
DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



DISK-ORIENTED DBMS



WHY NOT USE THE OS?

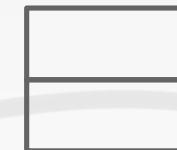
One can use memory mapping (**mmap**) to store the contents of a file into a process' address space.

The OS is responsible for moving data for moving the files' pages in and out of memory.

Virtual
Memory



Physical
Memory

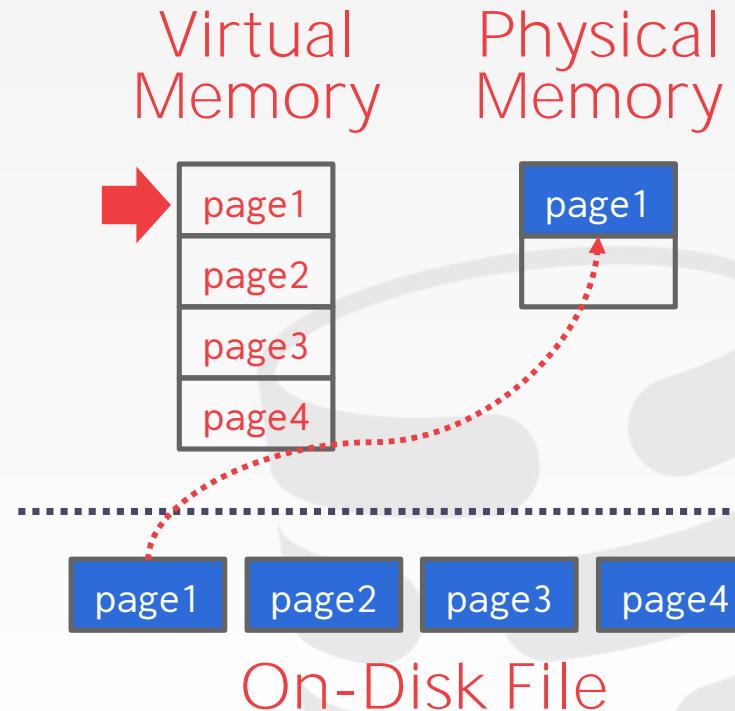


On-Disk File

WHY NOT USE THE OS?

One can use memory mapping (**mmap**) to store the contents of a file into a process' address space.

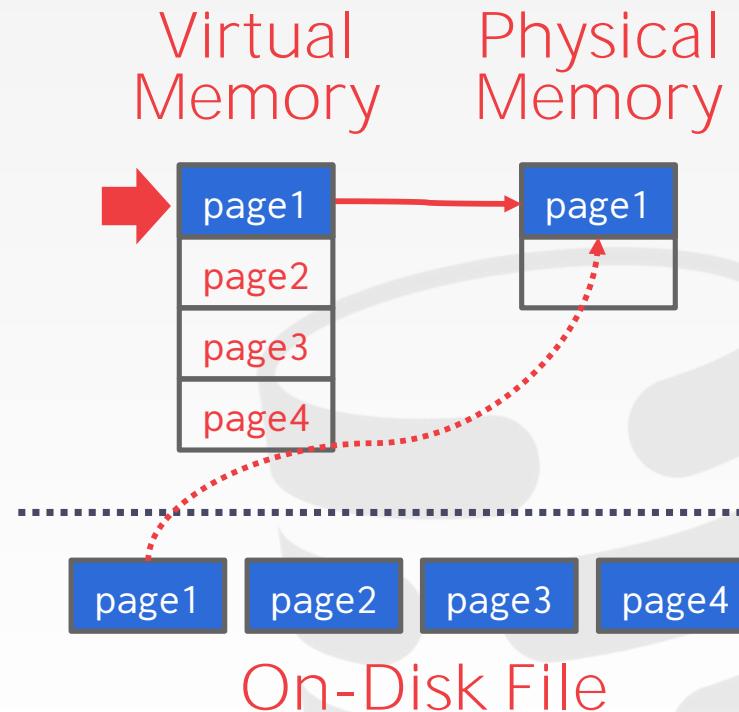
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use memory mapping (**mmap**) to store the contents of a file into a process' address space.

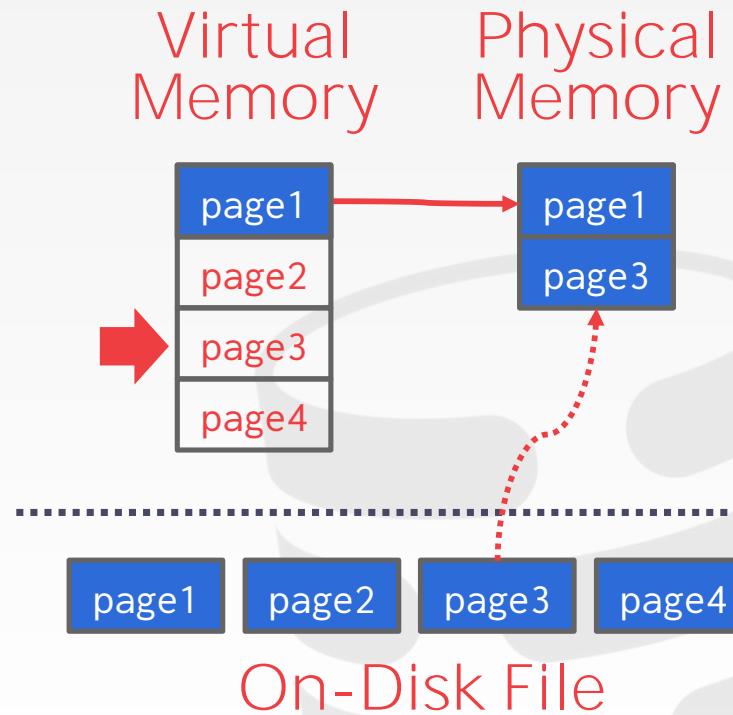
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use memory mapping (**mmap**) to store the contents of a file into a process' address space.

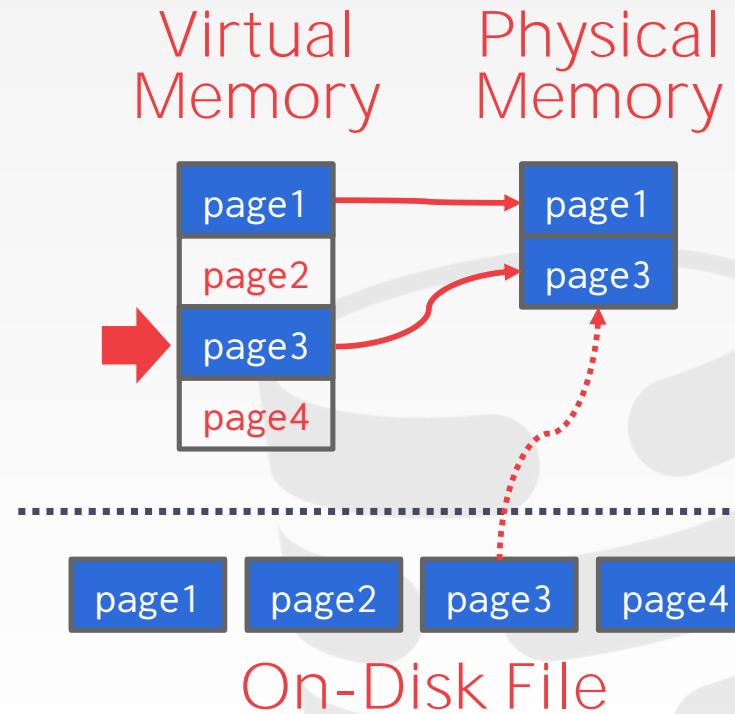
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use memory mapping (**mmap**) to store the contents of a file into a process' address space.

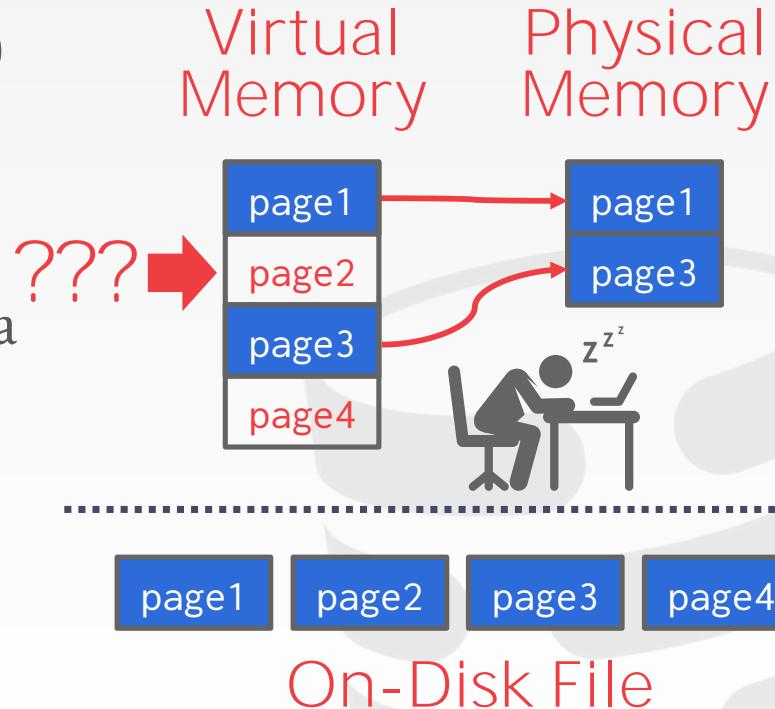
The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

One can use memory mapping (**mmap**) to store the contents of a file into a process' address space.

The OS is responsible for moving data for moving the files' pages in and out of memory.



WHY NOT USE THE OS?

What if we allow multiple threads to access the
mmap files to hide page fault stalls?

This works good enough for read-only access.
It is complicated when there are multiple writers...

WHY NOT USE THE OS?

There are some solutions to this problem:

- **madvise**: Tell the OS how you expect to read certain pages.
- **mlock**: Tell the OS that memory ranges cannot be paged out.
- **msync**: Tell the OS to flush memory ranges out to disk.

Full Usage



Partial Usage



WHY NOT USE THE OS?

DBMS (almost) always wants to control things itself and can do a better job at it.

- Flushing dirty pages to disk in the correct order.
- Specialized prefetching.
- Buffer replacement policy.
- Thread/process scheduling.

The OS is not your friend.



DATABASE STORAGE

Problem #1: How the DBMS represents the database in files on disk.

← Today

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.



TODAY'S AGENDA

File Storage

Page Layout

Tuple Layout



FILE STORAGE

The DBMS stores a database as one or more files on disk.

- The OS doesn't know anything about the contents of these files.

Early systems in the 1980s used custom filesystems on raw storage.

- Some "enterprise" DBMSs still support this.
- Most newer DBMSs do not do this.



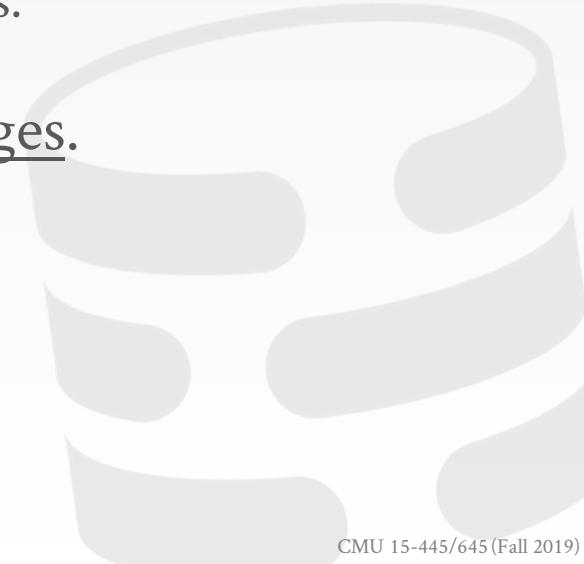
STORAGE MANAGER

The storage manager is responsible for maintaining a database's files.

- Some do their own scheduling for reads and writes to improve spatial and temporal locality of pages.

It organizes the files as a collection of pages.

- Tracks data read/written to pages.
- Tracks the available space.



DATABASE PAGES

A page is a fixed-size block of data.

- It can contain tuples, meta-data, indexes, log records...
- Most systems do not mix page types.
- Some systems require a page to be self-contained.

Each page is given a unique identifier.

- The DBMS uses an indirection layer to map page ids to physical locations.



DATABASE PAGES

There are three different notions of "pages" in a DBMS:

- Hardware Page (usually 4KB)
- OS Page (usually 4KB)
- Database Page (512B-16KB)

By hardware page, we mean at what level the device can guarantee a "failsafe write".

4KB



8KB



16KB



PAGE STORAGE ARCHITECTURE

Different DBMSs manage pages in files on disk in different ways.

- Heap File Organization
- Sequential / Sorted File Organization
- Hashing File Organization

At this point in the hierarchy we don't need to know anything about what is inside of the pages.



DATABASE HEAP

A heap file is an unordered collection of pages where tuples that are stored in random order.

- Create / Get / Write / Delete Page
- Must also support iterating over all pages.

Need meta-data to keep track of what pages exist and which ones have free space.

Two ways to represent a heap file:

- Linked List
- Page Directory

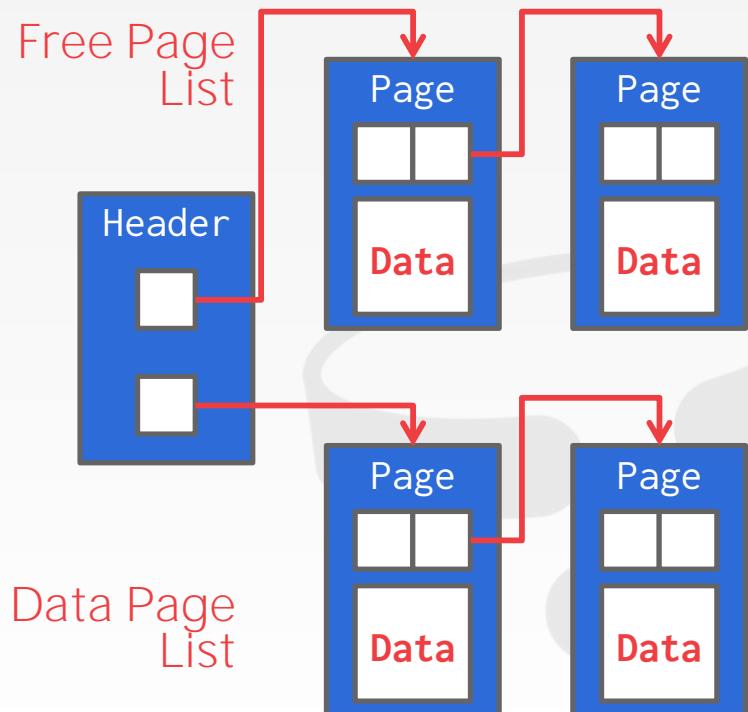


HEAP FILE: LINKED LIST

Maintain a header page at the beginning of the file that stores two pointers:

- HEAD of the free page list.
- HEAD of the data page list.

Each page keeps track of the number of free slots in itself.

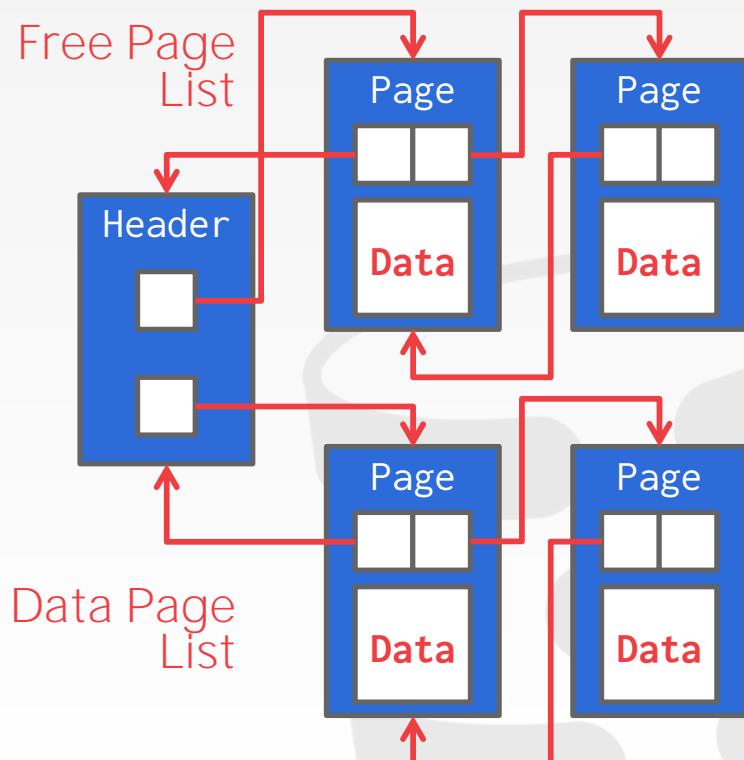


HEAP FILE: LINKED LIST

Maintain a header page at the beginning of the file that stores two pointers:

- HEAD of the free page list.
- HEAD of the data page list.

Each page keeps track of the number of free slots in itself.

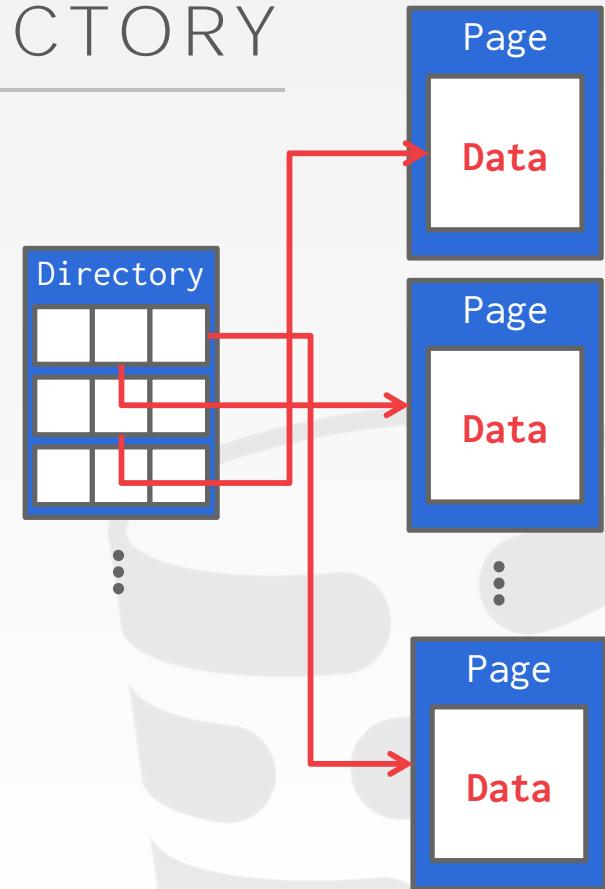


HEAP FILE: PAGE DIRECTORY

The DBMS maintains special pages that tracks the location of data pages in the database files.

The directory also records the number of free slots per page.

The DBMS has to make sure that the directory pages are in sync with the data pages.



TODAY'S AGENDA

File Storage

Page Layout

Tuple Layout

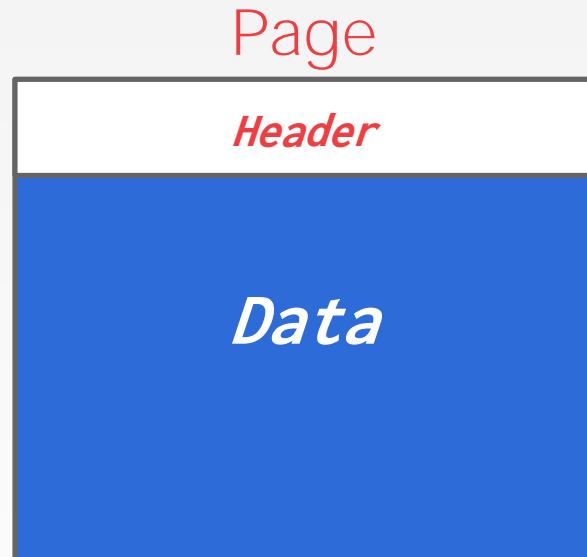


PAGE HEADER

Every page contains a header of meta-data about the page's contents.

- Page Size
- Checksum
- DBMS Version
- Transaction Visibility
- Compression Information

Some systems require pages to be self-contained (e.g., Oracle).



PAGE LAYOUT

For any page storage architecture, we now need to understand how to organize the data stored inside of the page.

→ We are still assuming that we are only storing tuples.

Two approaches:

- Tuple-oriented
- Log-structured



TUPLE STORAGE

How to store tuples in a page?

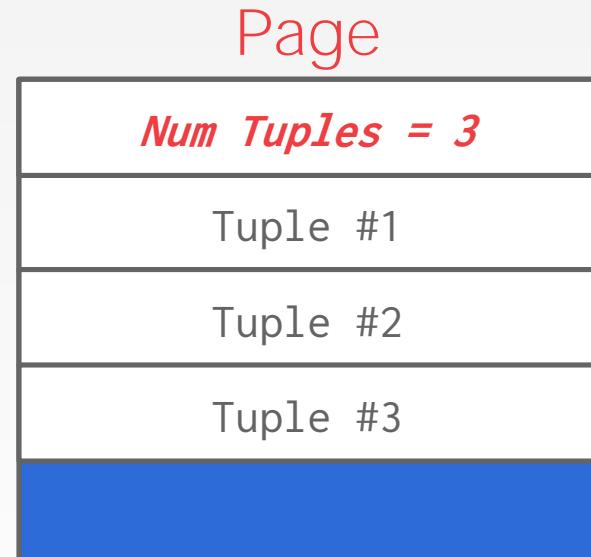
Page

Num Tuples = 0

TUPLE STORAGE

How to store tuples in a page?

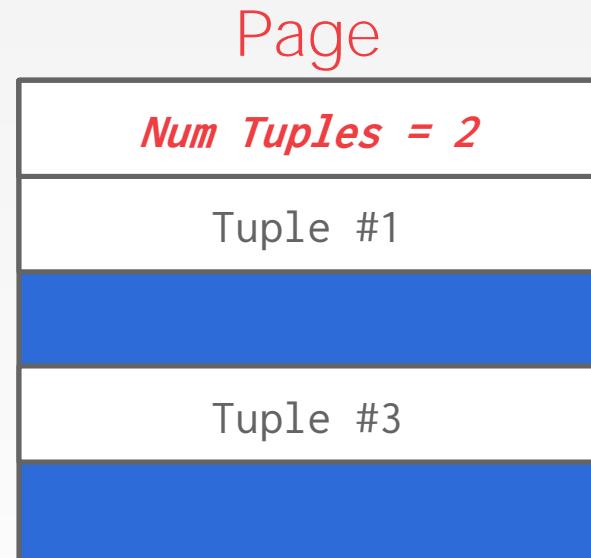
Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.



TUPLE STORAGE

How to store tuples in a page?

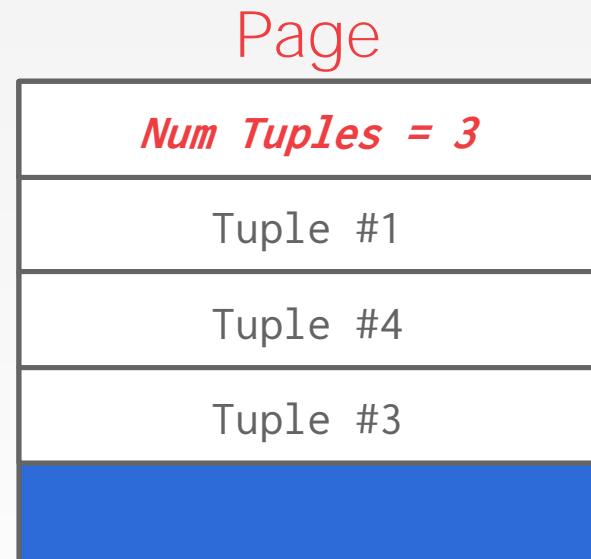
Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.
→ What happens if we delete a tuple?



TUPLE STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.
→ What happens if we delete a tuple?

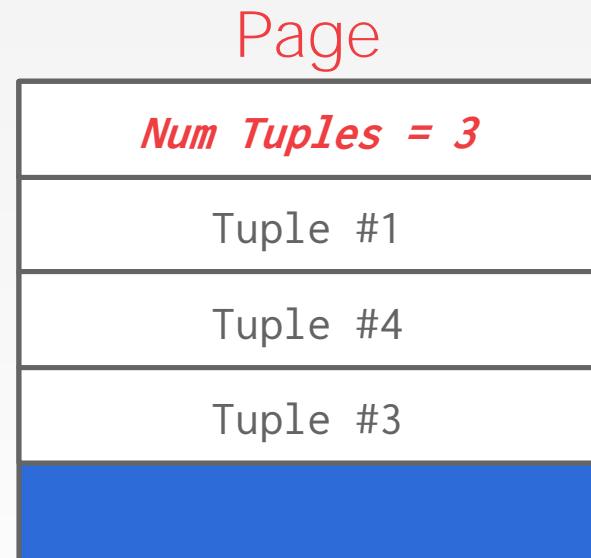


TUPLE STORAGE

How to store tuples in a page?

Strawman Idea: Keep track of the number of tuples in a page and then just append a new tuple to the end.

- What happens if we delete a tuple?
- What happens if we have a variable-length attribute?



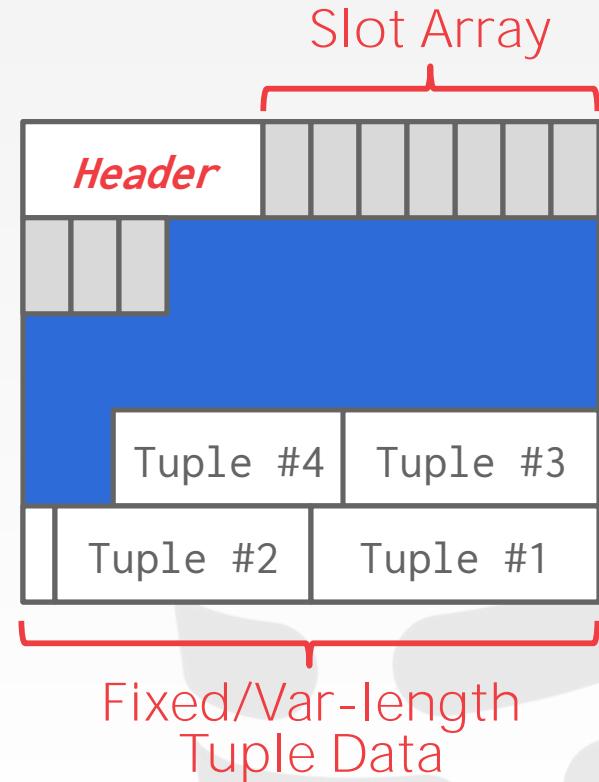
SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used.



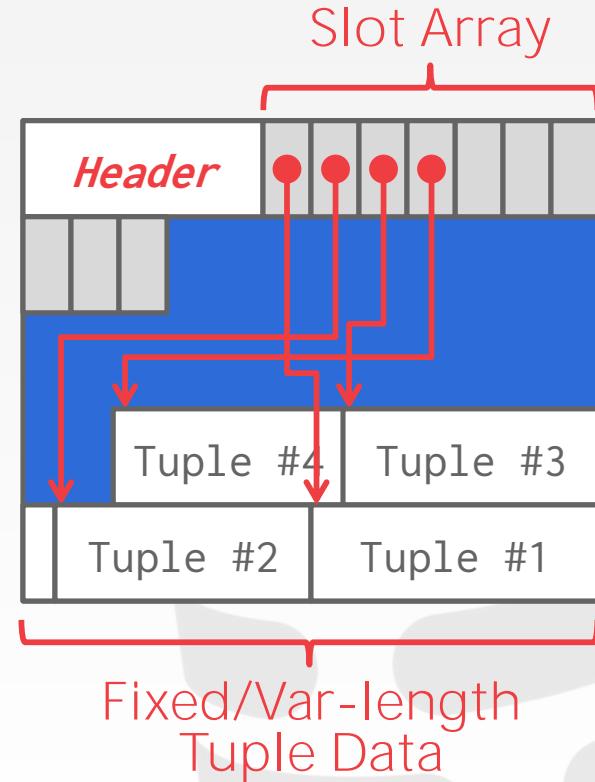
SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used.



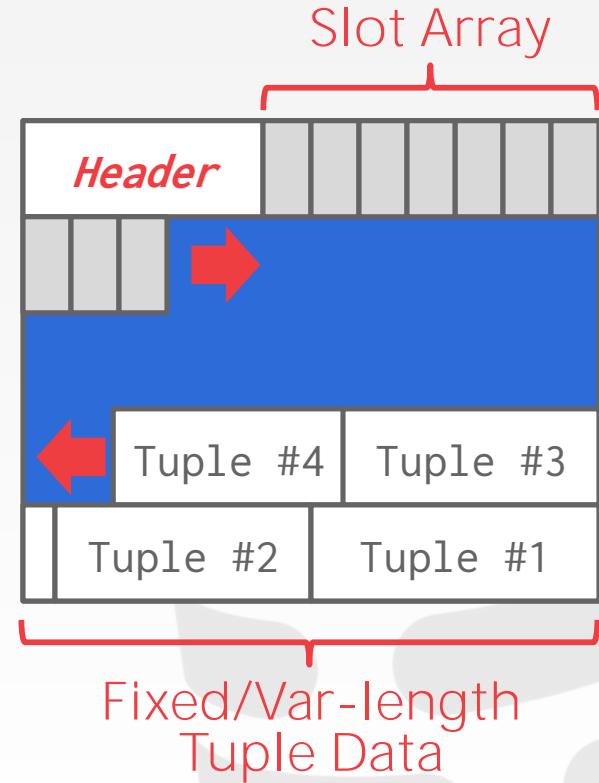
SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used.

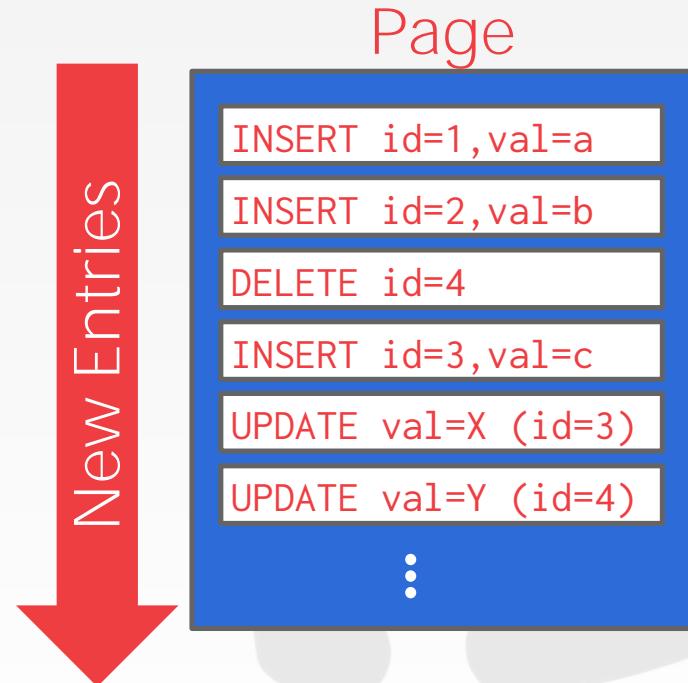


LOG-STRUCTURED FILE ORGANIZATION

Instead of storing tuples in pages, the DBMS only stores log records.

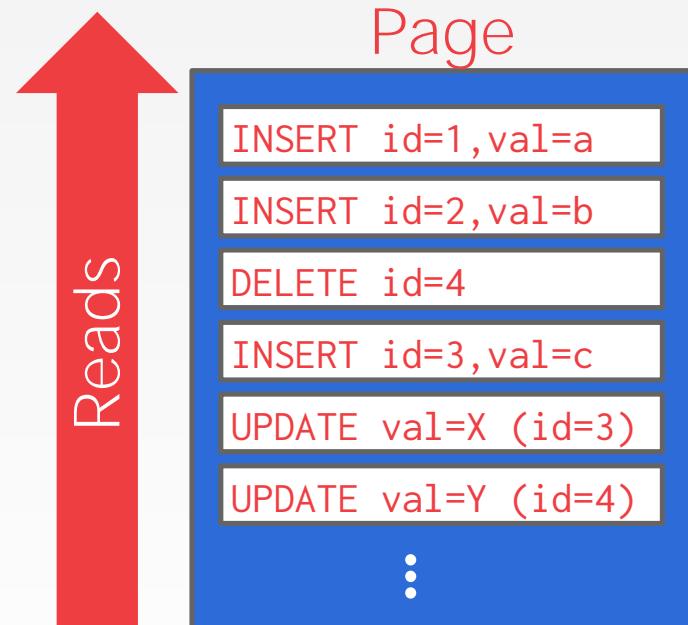
The system appends log records to the file of how the database was modified:

- Inserts store the entire tuple.
- Deletes mark the tuple as deleted.
- Updates contain the delta of just the attributes that were modified.



LOG-STRUCTURED FILE ORGANIZATION

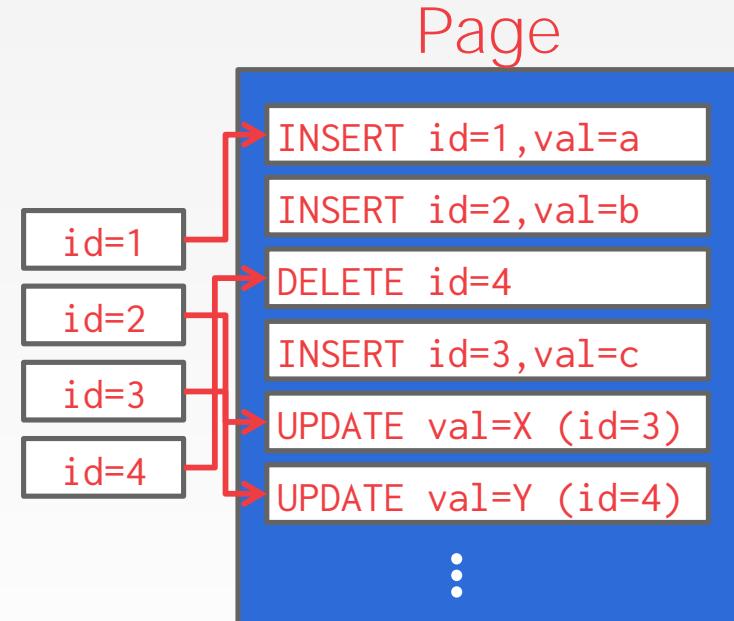
To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.



LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.



LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.

Periodically compact the log.

Page

```
id=1, val=a  
id=2, val=b  
id=3, val=X  
id=4, val=Y
```



RocksDB

TODAY'S AGENDA

File Storage

Page Layout

Tuple Layout



TUPLE LAYOUT

A tuple is essentially a sequence of bytes.

It's the job of the DBMS to interpret those bytes
into attribute types and values.



TUPLE HEADER

Each tuple is prefixed with a header that contains meta-data about it.

- Visibility info (concurrency control)
- Bit Map for **NULL** values.

We do not need to store meta-data about the schema.



TUPLE DATA

Attributes are typically stored in the order that you specify them when you create the table.

This is done for software engineering reasons.

We re-order attributes automatically in CMU's new DBMS...

Tuple

Header	a	b	c	d	e

```
CREATE TABLE foo (
    a INT PRIMARY KEY,
    b INT NOT NULL,
    c INT,
    d DOUBLE,
    e FLOAT
);
```

DENORMALIZED TUPLE DATA

Can physically *denormalize* (e.g., "pre join") related tuples and store them together in the same page.

- Potentially reduces the amount of I/O for common workload patterns.
- Can make updates more expensive.

```
CREATE TABLE foo (
    a INT PRIMARY KEY,
    b INT NOT NULL,
);
CREATE TABLE bar (
    c INT PRIMARY KEY,
    a INT
    REFERENCES foo (a),
);
```

The diagram illustrates the creation of two tables, foo and bar. The first table, foo, has columns a (INT, PRIMARY KEY) and b (INT, NOT NULL). The second table, bar, has columns c (INT, PRIMARY KEY) and a (INT). A red box encloses the entire definition of table foo. A red arrow points from the 'REFERENCES' keyword in the definition of table bar to the 'a' column in the definition of table foo, indicating a foreign key relationship.

DENORMALIZED TUPLE DATA

Can physically *denormalize* (e.g., "pre join") related tuples and store them together in the same page.

- Potentially reduces the amount of I/O for common workload patterns.
- Can make updates more expensive.

foo

<i>Header</i>	a	b
---------------	---	---

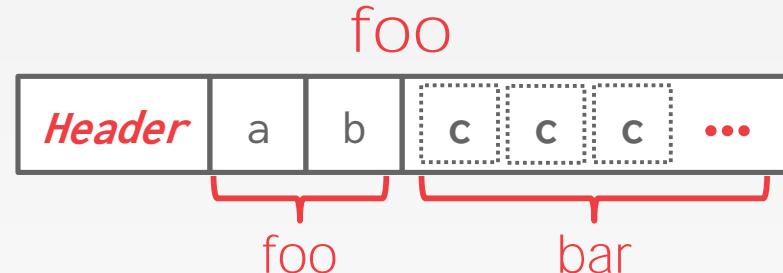
bar

<i>Header</i>	c	a
<i>Header</i>	c	a
<i>Header</i>	c	a

DENORMALIZED TUPLE DATA

Can physically *denormalize* (e.g., "pre join") related tuples and store them together in the same page.

- Potentially reduces the amount of I/O for common workload patterns.
- Can make updates more expensive.



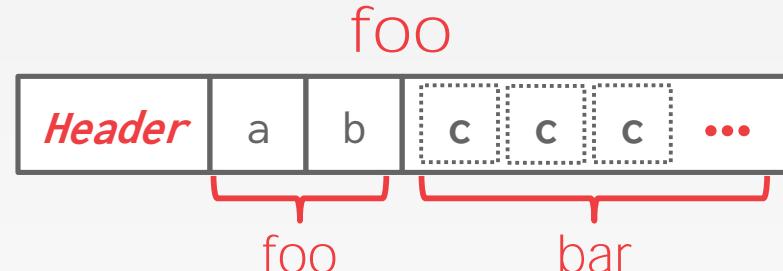
DENORMALIZED TUPLE DATA

Can physically *denormalize* (e.g., "pre join") related tuples and store them together in the same page.

- Potentially reduces the amount of I/O for common workload patterns.
- Can make updates more expensive.

Not a new idea.

- IBM System R did this in the 1970s.
- Several NoSQL DBMSs do this without calling it physical denormalization.



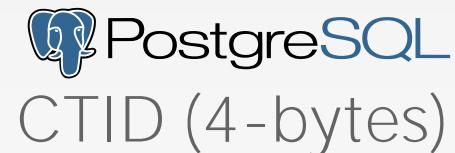
RECORD IDS

The DBMS needs a way to keep track of individual tuples.

Each tuple is assigned a unique record identifier.

- Most common: **page_id + offset/slot**
- Can also contain file location info.

An application cannot rely on these ids to mean anything.



CONCLUSION

Database is organized in pages.

Different ways to track pages.

Different ways to store pages.

Different ways to store tuples.



NEXT CLASS

Value Representation
Storage Models



04

Database Storage – Part II



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #1 is due September 11th @ 11:59pm

Project #1 will be released on September 11th



UPCOMING DATABASE EVENTS

SalesForce Talk

- Friday Sep 13th @ 12:00pm
- CIC 4th Floor



Impira Talk

- Monday Sep 16th @ 4:30pm
- GHC 8102



Vertica Talk

- Monday Sep 23rd @ 4:30pm
- GHC 8102



DISK-ORIENTED ARCHITECTURE

The DBMS assumes that the primary storage location of the database is on non-volatile disk.

The DBMS's components manage the movement of data between non-volatile and volatile storage.



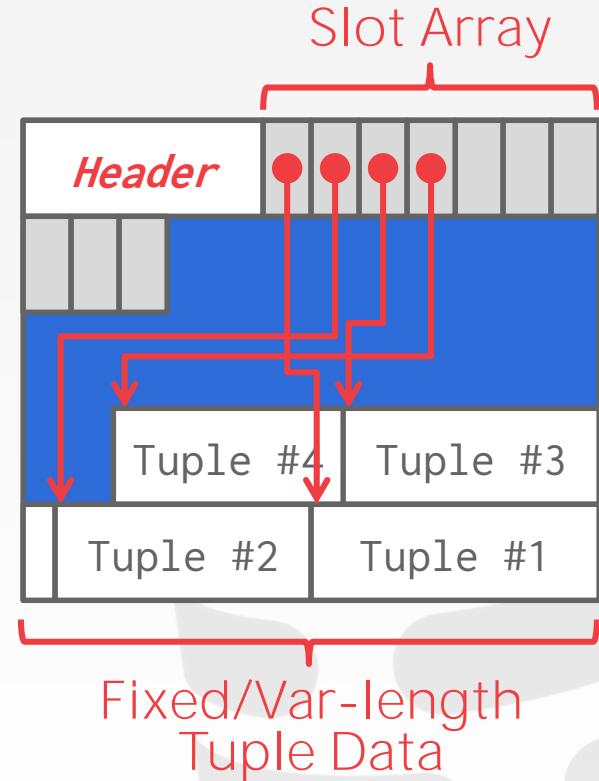
SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used.



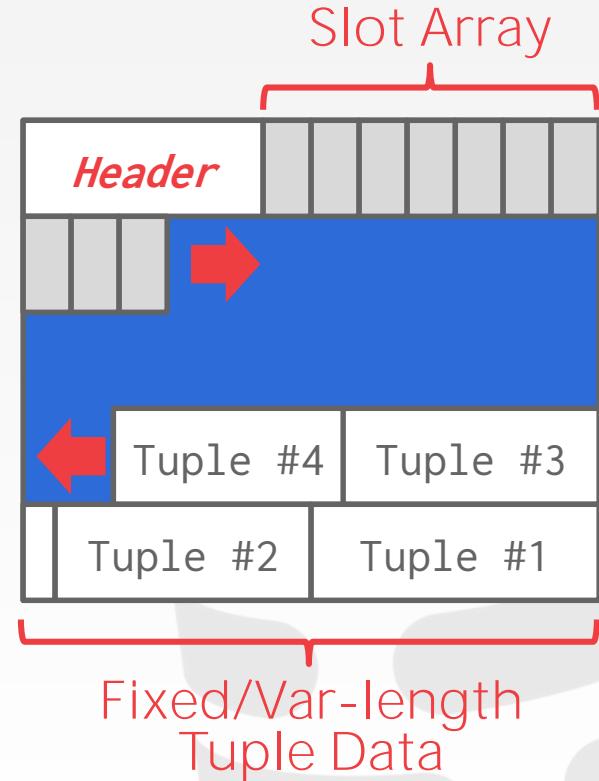
SLOTTED PAGES

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used.

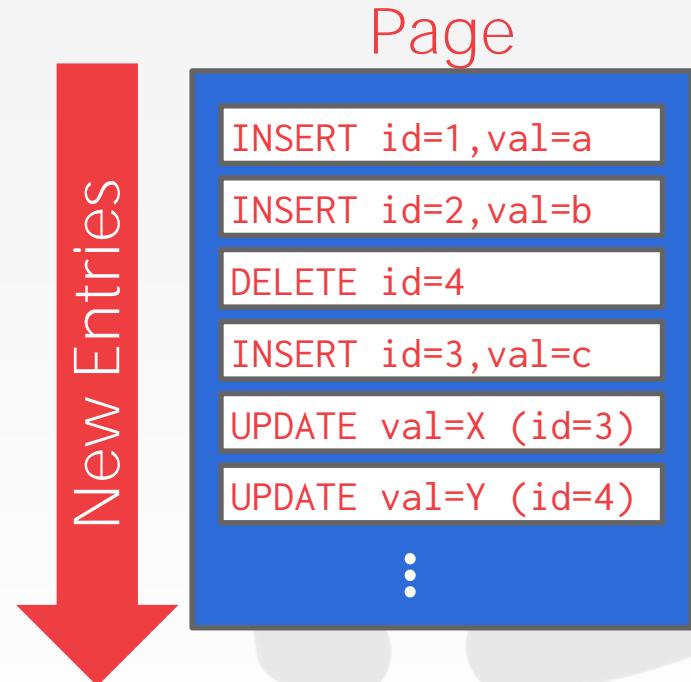


LOG-STRUCTURED FILE ORGANIZATION

Instead of storing tuples in pages, the DBMS only stores log records.

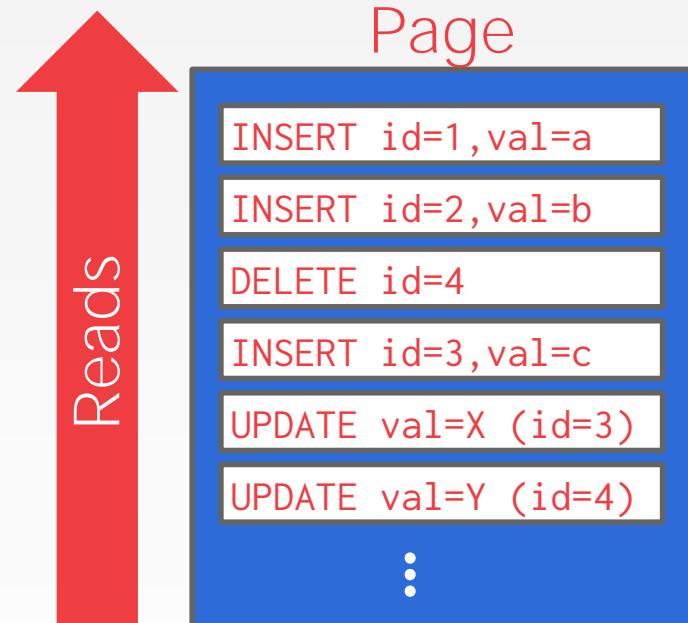
The system appends log records to the file of how the database was modified:

- Inserts store the entire tuple.
- Deletes mark the tuple as deleted.
- Updates contain the delta of just the attributes that were modified.



LOG-STRUCTURED FILE ORGANIZATION

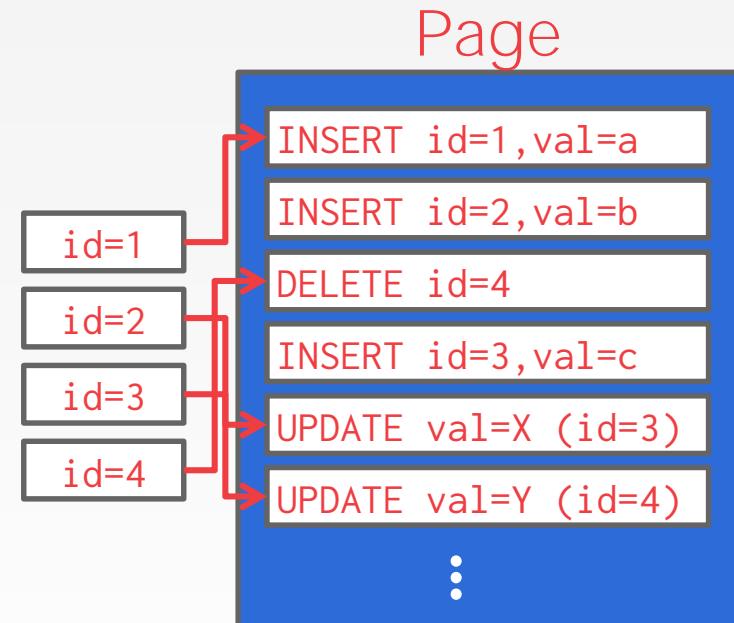
To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.



LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.



LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.

Periodically compact the log.

Page

```
id=1, val=a  
id=2, val=b  
id=3, val=X  
id=4, val=Y
```

LOG-STRUCTURED FILE ORGANIZATION

To read a record, the DBMS scans the log backwards and "recreates" the tuple to find what it needs.

Build indexes to allow it to jump to locations in the log.

Periodically compact the log.

Page

```
id=1, val=a  
id=2, val=b  
id=3, val=X  
id=4, val=Y
```



levelDB



RocksDB

TODAY'S AGENDA

Data Representation
System Catalogs
Storage Models

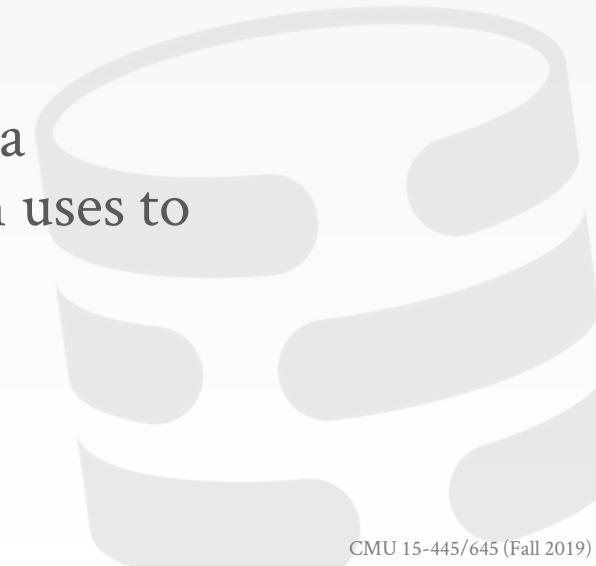


TUPLE STORAGE

A tuple is essentially a sequence of bytes.

It's the job of the DBMS to interpret those bytes into attribute types and values.

The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.



DATA REPRESENTATION

INTEGER/BIGINT/SMALLINT/TINYINT

→ C/C++ Representation

FLOAT/REAL vs. **NUMERIC/DECIMAL**

→ IEEE-754 Standard / Fixed-point Decimals

VARCHAR/VARBINARY/TEXT/BLOB

→ Header with length, followed by data bytes.

TIME/DATE/TIMESTAMP

→ 32/64-bit integer of (micro)seconds since Unix epoch



VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the "native" C/C++ types.

→ Examples: **FLOAT, REAL/DOUBLE**

Store directly as specified by [IEEE-754](#).

Typically faster than arbitrary precision numbers but can have rounding errors...



VARIABLE PRECISION NUMBERS

Rounding Example

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %f\n", x+y);
    printf("0.3 = %f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

VARIABLE PRECISION NUMBERS

Rounding Example

```
#include <stdio.h>

in #include <stdio.h>

int main(int argc, char* argv[]) {
    float x = 0.1;
    float y = 0.2;
    printf("x+y = %.20f\n", x+y);
    printf("0.3 = %.20f\n", 0.3);
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

```
x+y = 0.3000001192092895508
0.3 = 0.2999999999999998890
```

FIXED PRECISION NUMBERS

Numeric data types with arbitrary precision and scale. Used when round errors are unacceptable.

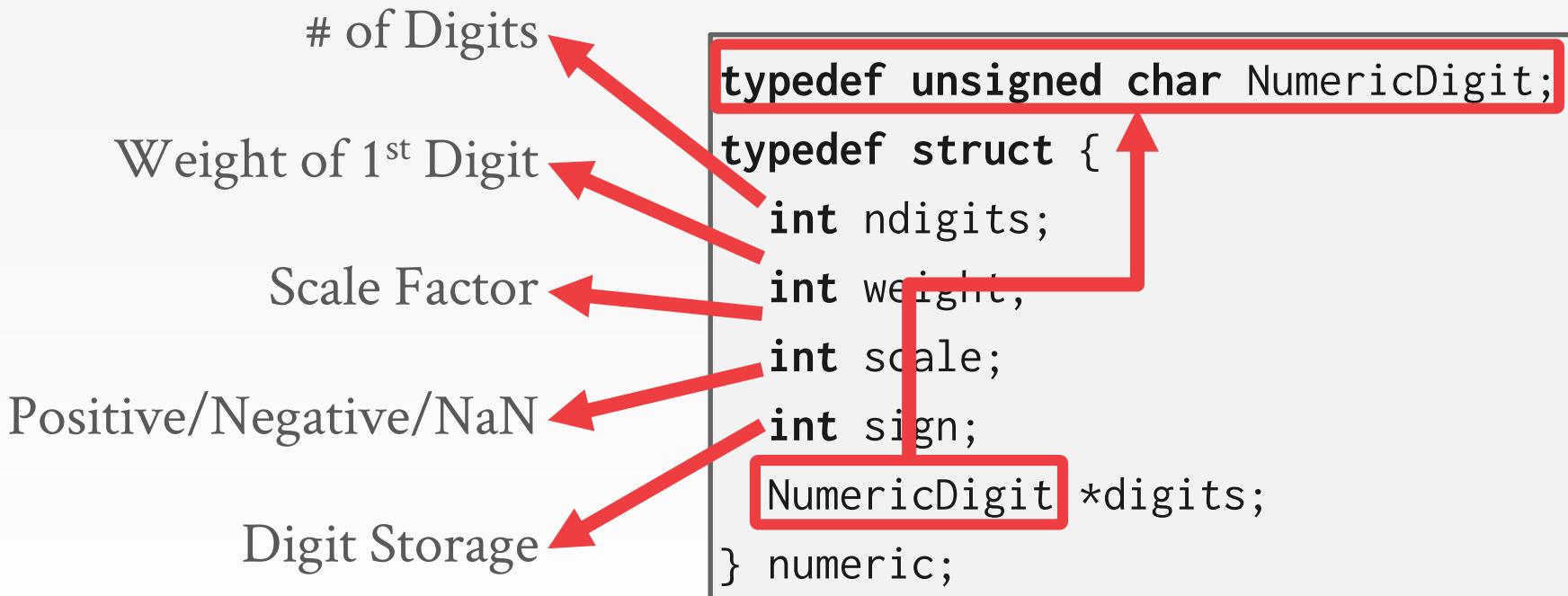
→ Example: **NUMERIC, DECIMAL**

Typically stored in an exact, variable-length binary representation with additional meta-data.

→ Like a **VARCHAR** but not stored as a string

Demo: Postgres, SQL Server, Oracle

POSTGRES: NUMERIC



Weight of Sca Positive/Negat Digi

```

/*
 * add_var() -
 *
 * Full version of add functionality on variable level (handling signs).
 * result might point to one of the operands too without danger.
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /*
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * -----
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /*
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * -----
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /*
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * -----
                     */
                    if (sub_abs(var2, var1, result) != 0)
                        return -1;
                    result->sign = NUMERIC_NEG;
                    break;
            }
        }
    }
}

```

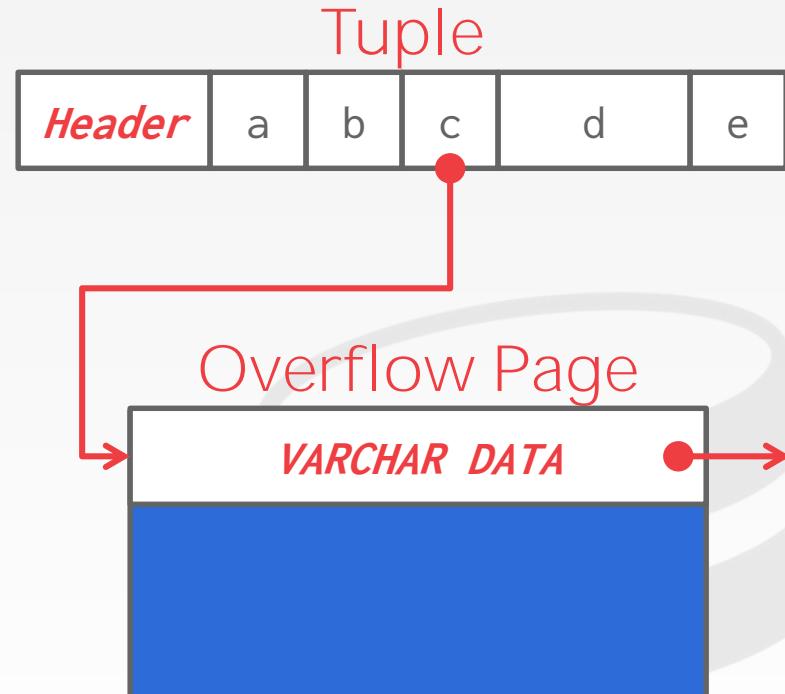
NumericDigit;

LARGE VALUES

Most DBMSs don't allow a tuple to exceed the size of a single page.

To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

- Postgres: TOAST (>2KB)
- MySQL: Overflow (> $\frac{1}{2}$ size of page)
- SQL Server: Overflow (>size of page)



EXTERNAL VALUE STORAGE

Some systems allow you to store a really large value in an external file.

Treated as a **BLOB** type.

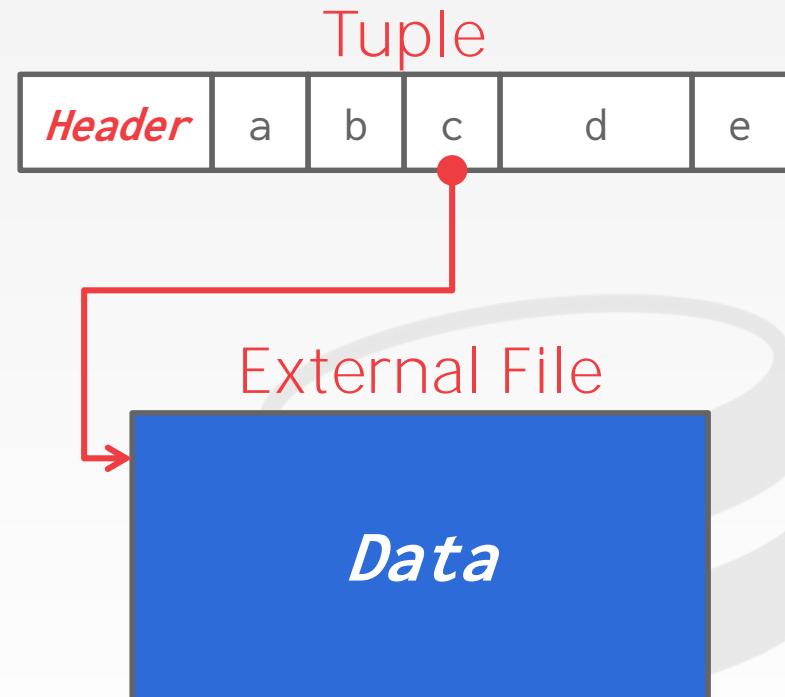
→ Oracle: **BFILE** data type

→ Microsoft: **FILESTREAM** data type

The DBMS cannot manipulate the contents of an external file.

→ No durability protections.

→ No transaction protections.



EXTERNAL VALUE

Some systems allow you to store a really large value in an external file.

Treated as a **BLOB** type.

→ Oracle: **BFILE** data type

→ Microsoft: **FILESTREAM** data type

The DBMS cannot manipulate the contents of an external file.

- No durability protections.
- No transaction protections.

To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?

Russell Sears², Catharine van Ingen¹, Jim Gray¹
 1: Microsoft Research, 2: University of California at Berkeley
 sears@cs.berkeley.edu, vaningen@microsoft.com, gray@microsoft.com
 MSR-TR-2006-45
 April 2006 Revised June 2006

Abstract

Application designers must decide whether to store large objects (BLOBs) in a filesystem or in a database. Generally, this decision is based on factors such as application simplicity or manageability. Often, system performance affects these factors.

Folklore tells us that databases efficiently handle large numbers of small objects, while filesystems are more efficient for large objects. Where is the break-even point? When is accessing a BLOB stored as a file cheaper than accessing a BLOB stored as a database record?

Of course, this depends on the particular filesystem, database system, and workload in question. This study shows that when comparing the NTFS file system and SQL Server 2005 database system on a *create, (read, replace), delete* workload, BLOBS smaller than 256KB are more efficiently handled by SQL Server, while NTFS is more efficient BLOBS larger than 1MB. Of course, this break-even point will vary among different database systems, filesystems, and workloads.

By measuring the performance of a storage server workload typical of web applications which use get/put protocols such as WebDAV [WebDAV], we found that the break-even point depends on many factors. However, our experiments suggest that storage age, the ratio of bytes in deleted or replaced objects to bytes in live objects, is dominant. As storage age increases, fragmentation tends to increase. The filesystem we study has better fragmentation control than the database we used, suggesting the database system would benefit from incorporating ideas from filesystem architecture. Conversely, filesystem performance may be improved by using database techniques to handle small files.

Surprisingly, for these studies, when average object size is held constant, the distribution of object sizes did not significantly affect performance. We also found that, in addition to low percentage free space, a low ratio of free space to average object size leads to fragmentation and performance degradation.

1. Introduction

Application data objects are getting larger as digital media becomes ubiquitous. Furthermore, the increasing popularity of web services and other network applications means that systems that once managed static archives of "finished" objects now manage frequently modified versions of application data as it is being created and updated. Rather than updating these objects, the archive either stores multiple versions of the objects (the V of WebDAV stands for "versioning"), or simply does wholesale replacement (as in SharePoint Team Services [SharePoint]).

Application designers have the choice of storing large objects as files in the filesystem, as BLOBs (binary large objects) in a database, or as a combination of both. Only folklore is available regarding the tradeoffs after the design decision is based on which technology the designer knows best. Most designers will tell you that a database is probably best for small binary objects and that files are best for large objects. But, what is the break-even point? What are the tradeoffs?

This article characterizes the performance of an abstracted write-intensive web application that deals with relatively large objects. Two versions of the system are compared; one uses a relational database to store large objects, while the other version stores the objects as files in the filesystem. We measure how performance changes over time as the storage becomes fragmented. The article concludes by describing and quantifying the factors that a designer should consider when picking a storage system. It also suggests filesystem and database improvements for large object support.

One surprising (to us at least) conclusion of our work is that storage fragmentation is the main determinant of the break-even point in the tradeoff. Therefore, much of our work and much of this article focuses on storage fragmentation issues. In essence, filesystems seem to have better fragmentation handling than databases and this drives the break-even point down from about 1MB to about 256KB.

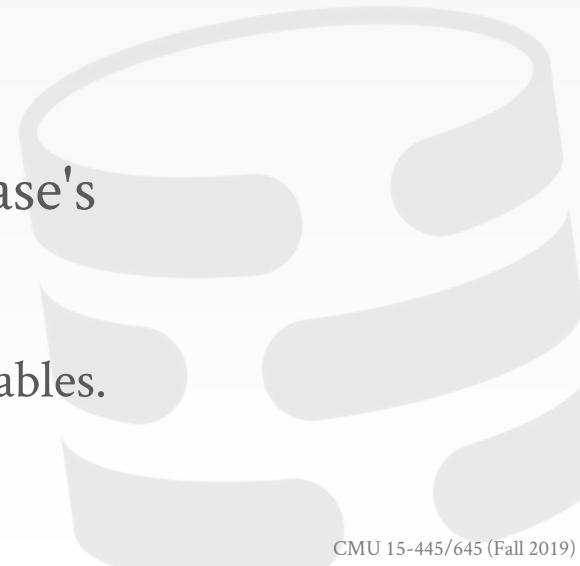
SYSTEM CATALOGS

A DBMS stores meta-data about databases in its internal catalogs.

- Tables, columns, indexes, views
- Users, permissions
- Internal statistics

Almost every DBMS stores their a database's catalog in itself.

- Wrap object abstraction around tuples.
- Specialized code for "bootstrapping" catalog tables.



SYSTEM CATALOGS

You can query the DBMS's internal **INFORMATION_SCHEMA** catalog to get info about the database.

→ ANSI standard set of read-only views that provide info about all of the tables, views, columns, and procedures in a database

DBMSs also have non-standard shortcuts to retrieve this information.

ACCESSING TABLE SCHEMA

List all the tables in the current database:

```
SELECT *                                SQL-92  
FROM INFORMATION_SCHEMA.TABLES  
WHERE table_catalog = '<db name>';
```

```
\d;                                     Postgres
```

```
SHOW TABLES;                            MySQL
```

```
.tables;                                 SQLite
```

ACCESSING TABLE SCHEMA

List all the tables in the student table:

```
SELECT *                                SQL-92  
FROM INFORMATION_SCHEMA.TABLES  
WHERE table_name = 'student'
```

```
\d student;                            Postgres
```

```
DESCRIBE student;                      MySQL
```

```
.schema student;                      SQLite
```

OBSERVATION

The relational model does **not** specify that we have to store all of a tuple's attributes together in a single page.

This may **not** actually be the best layout for some workloads...



WIKIPEDIA EXAMPLE

```
CREATE TABLE useracct (
    userID INT PRIMARY KEY,
    userName VARCHAR UNIQUE,
    :
);
```

```
CREATE TABLE pages (
    pageID INT PRIMARY KEY,
    title VARCHAR UNIQUE,
    latest INT
    ↗ REFERENCES revisions (revID),
);
```

```
CREATE TABLE revisions (
    revID INT PRIMARY KEY,
    userID INT REFERENCES useracct (userID),
    pageID INT REFERENCES pages (pageID),
    content TEXT,
    updated DATETIME
);
```

OLTP

On-line Transaction Processing:

→ Simple queries that read/update a small amount of data that is related to a single entity in the database.

This is usually the kind of application that people build first.

```
SELECT P.* , R.*  
FROM pages AS P  
INNER JOIN revisions AS R  
ON P.latest = R.revID  
WHERE P.pageID = ?
```

```
UPDATE useracct  
    SET lastLogin = NOW(),  
        hostname = ?  
WHERE userID = ?
```

```
INSERT INTO revisions  
VALUES (?, ?, ..., ?)
```

OLAP

On-line Analytical Processing:

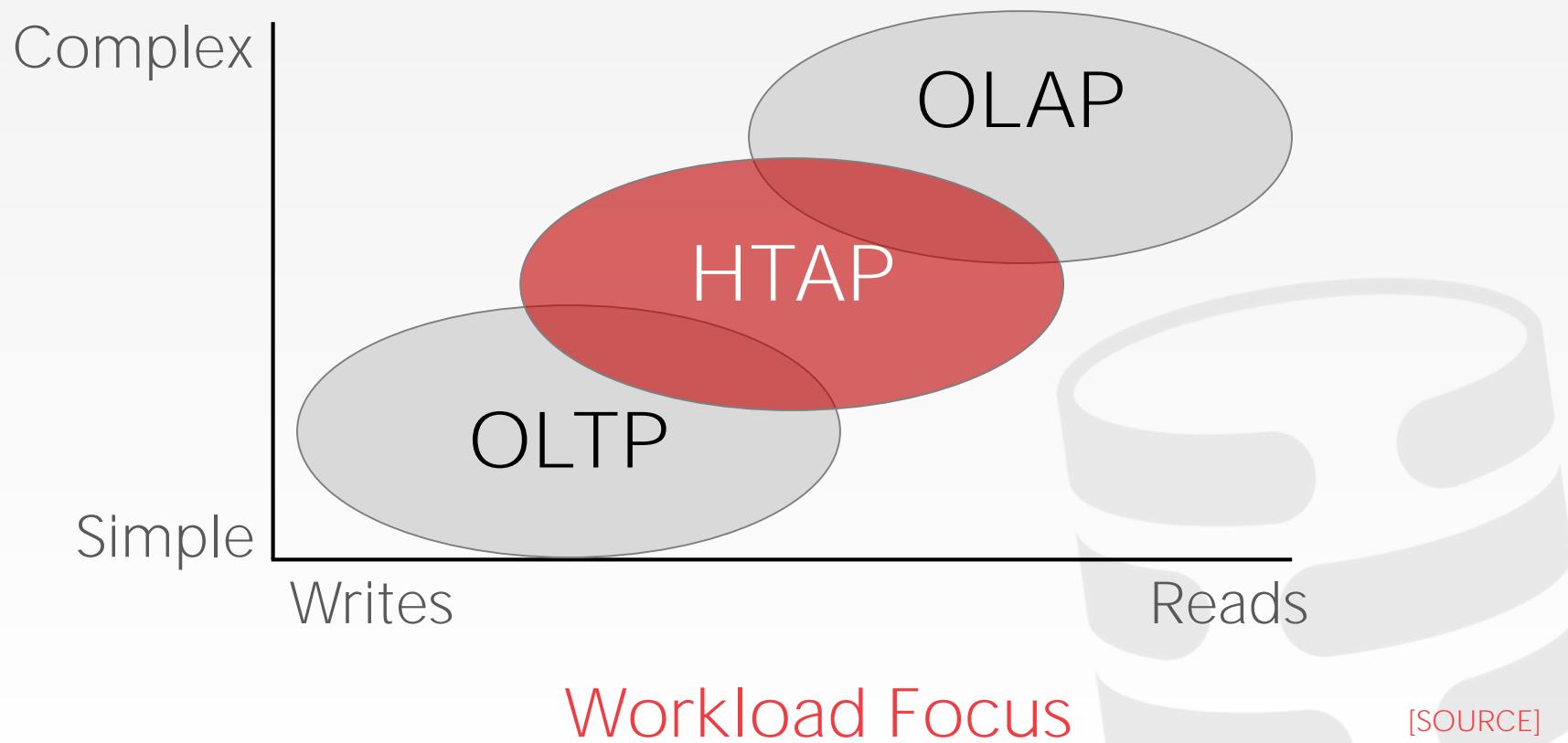
→ Complex queries that read large portions of the database spanning multiple entities.

You execute these workloads on the data you have collected from your OLTP application(s).

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM  
              U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY  
       EXTRACT(month FROM U.lastLogin)
```

Operation Complexity

WORKLOAD CHARACTERIZATION

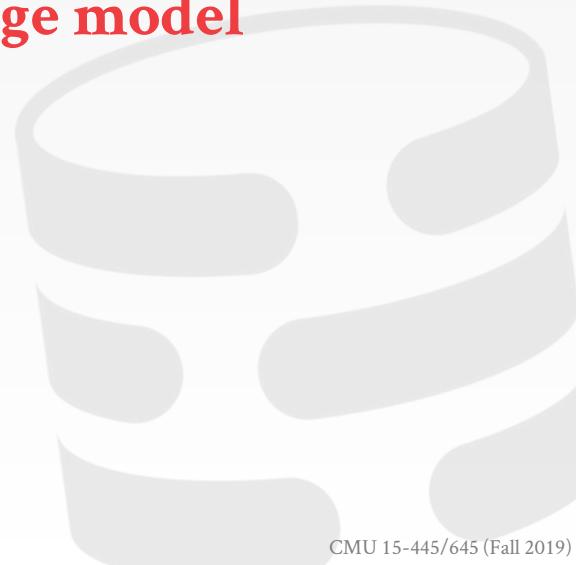


[SOURCE]

DATA STORAGE MODELS

The DBMS can store tuples in different ways that are better for either OLTP or OLAP workloads.

We have been assuming the **n-ary storage model** (aka "row storage") so far this semester.



N-ARY STORAGE MODEL (NSM)

The DBMS stores all attributes for a single tuple contiguously in a page.

Ideal for OLTP workloads where queries tend to operate only on an individual entity and insert-heavy workloads.



N-ARY STORAGE MODEL (NSM)

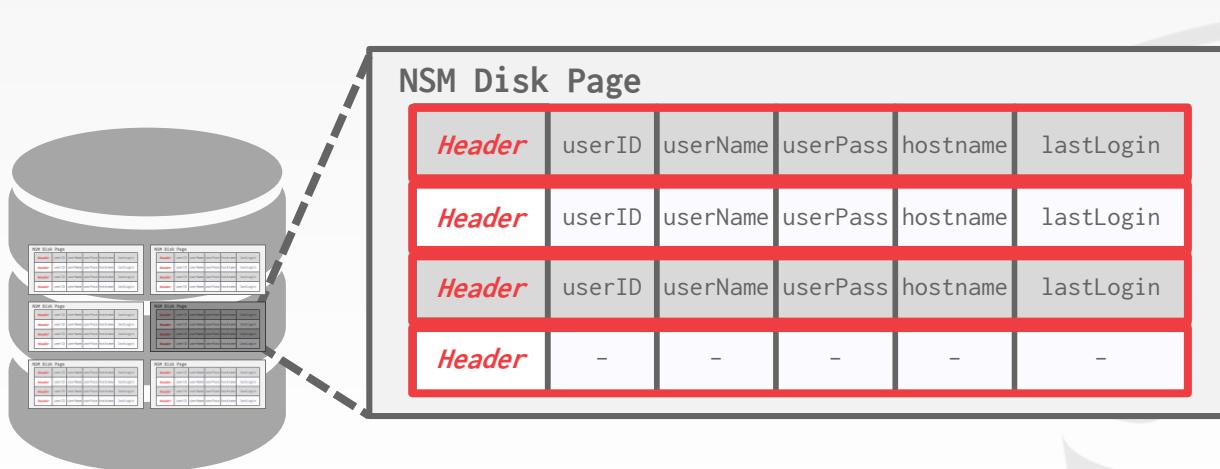
The DBMS stores all attributes for a single tuple contiguously in a page.



<i>Header</i>	userID	userName	userPass	hostname	lastLogin	← Tuple #1
<i>Header</i>	userID	userName	userPass	hostname	lastLogin	← Tuple #2
<i>Header</i>	userID	userName	userPass	hostname	lastLogin	← Tuple #3
<i>Header</i>	-	-	-	-	-	← Tuple #4

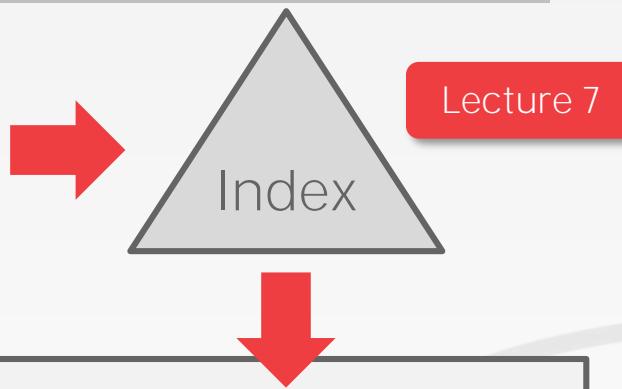
N-ARY STORAGE MODEL (NSM)

The DBMS stores all attributes for a single tuple contiguously in a page.

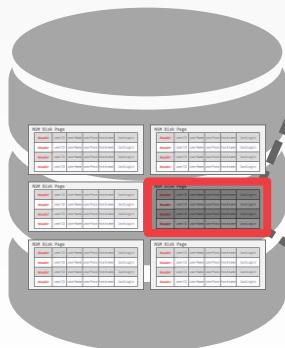


N-ARY STORAGE MODEL (NSM)

```
SELECT * FROM useracct  
WHERE userName = ?  
AND userPass = ?
```



Lecture 7



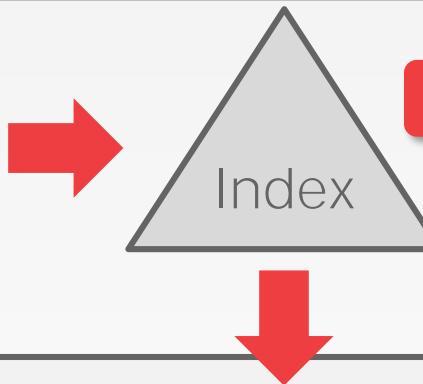
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	userID	userName	userPass	hostname	lastLogin
Header	-	-	-	-	-

N-ARY STORAGE MODEL (NSM)

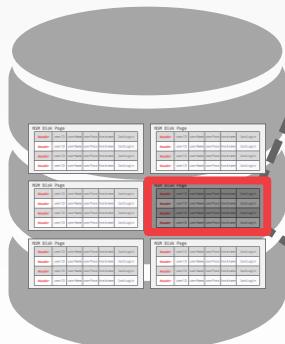
```
SELECT * FROM useracct
WHERE userName = ?
AND userPass = ?
```

```
INSERT INTO useracct
VALUES (?, ?, ...?)
```

Lecture 7



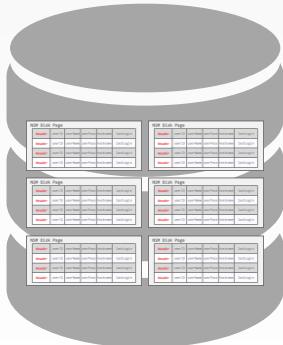
NSM Disk Page



<i>Header</i>	userID	userName	userPass	hostname	lastLogin
<i>Header</i>	userID	userName	userPass	hostname	lastLogin
<i>Header</i>	userID	userName	userPass	hostname	lastLogin
<i>Header</i>	userID	userName	userPass	hostname	lastLogin

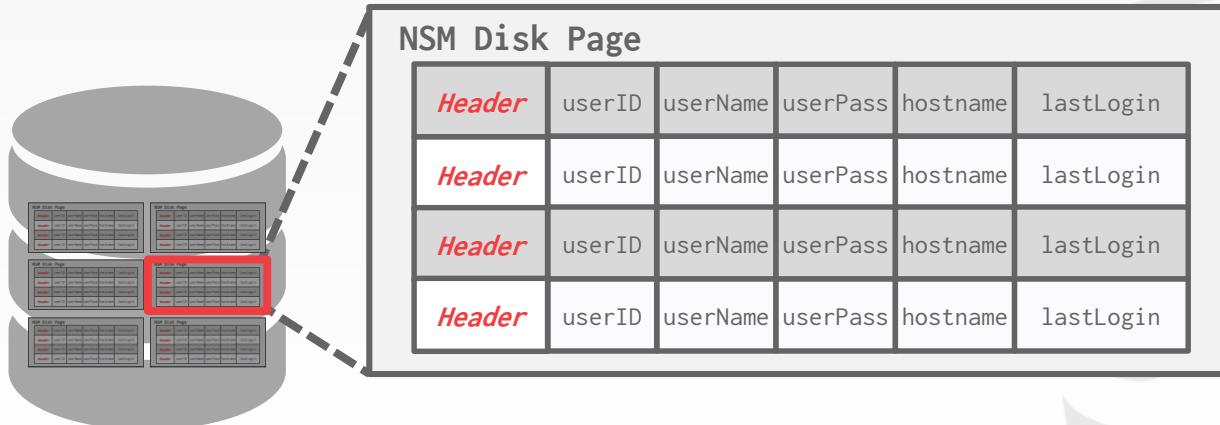
N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



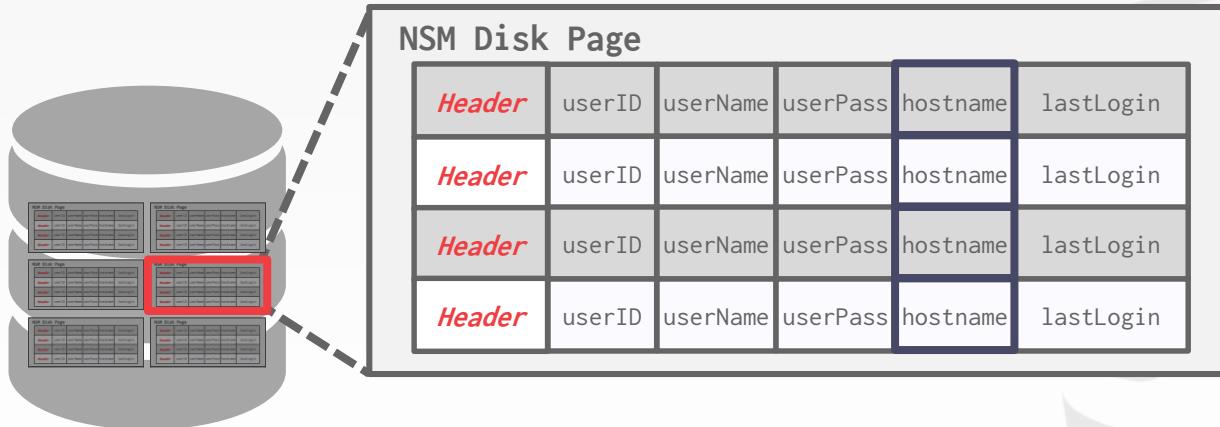
N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



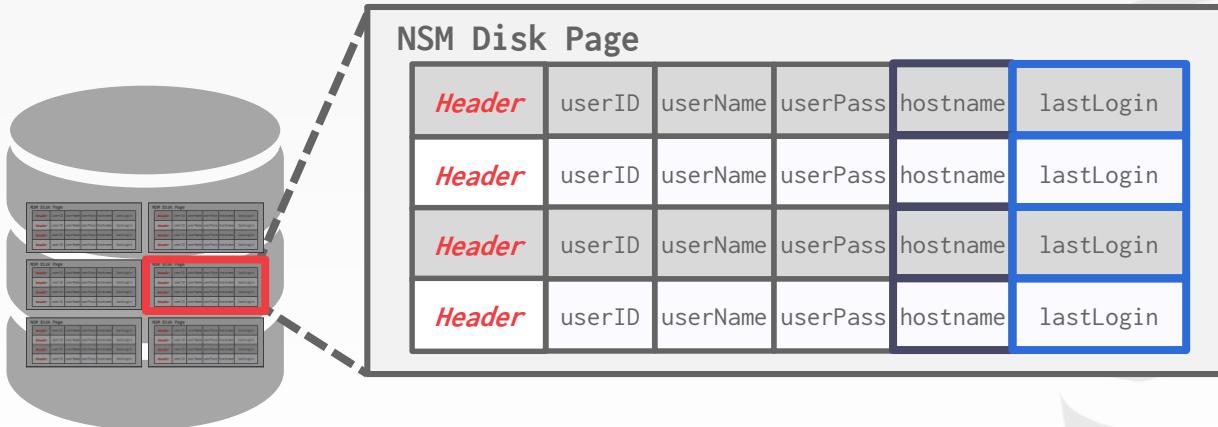
N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



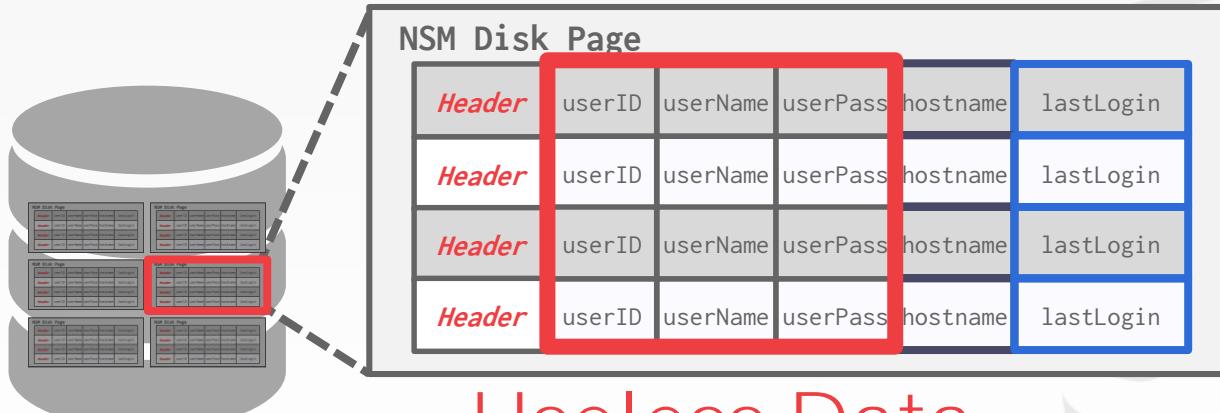
N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



N-ARY STORAGE MODEL (NSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



Useless Data

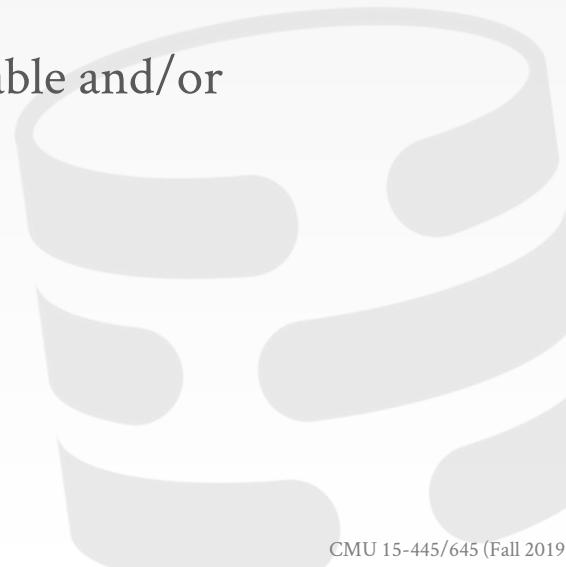
N-ARY STORAGE MODEL

Advantages

- Fast inserts, updates, and deletes.
- Good for queries that need the entire tuple.

Disadvantages

- Not good for scanning large portions of the table and/or a subset of the attributes.



DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute for all tuples contiguously in a page.
→ Also known as a "column store".

Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.

DECOMPOSITION STORAGE MODEL (DSM)

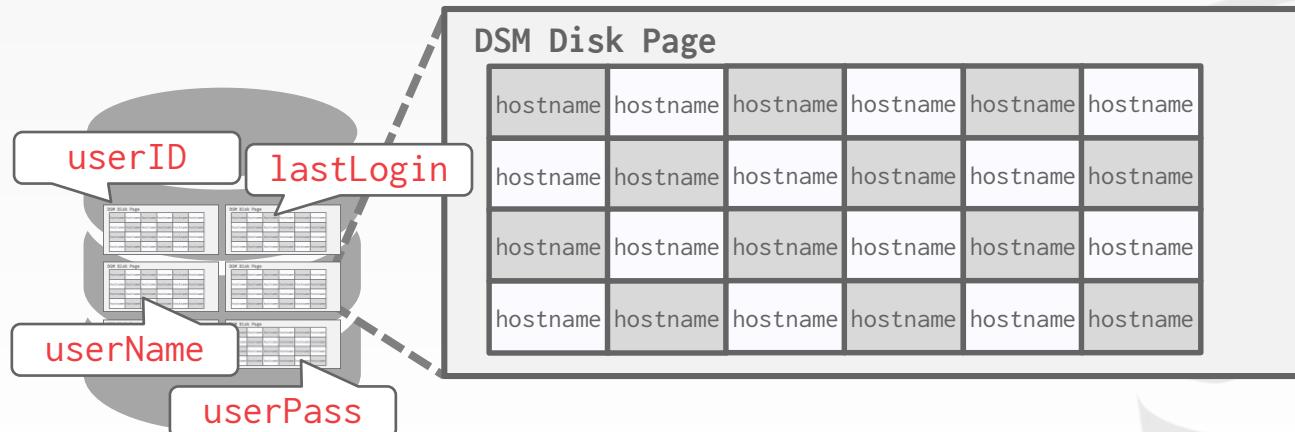
The DBMS stores the values of a single attribute for all tuples contiguously in a page.
→ Also known as a "column store".



<i>Header</i>	userID	userName	userPass	hostname	lastLogin
<i>Header</i>	userID	userName	userPass	hostname	lastLogin
<i>Header</i>	userID	userName	userPass	hostname	lastLogin
<i>Header</i>	userID	userName	userPass	hostname	lastLogin

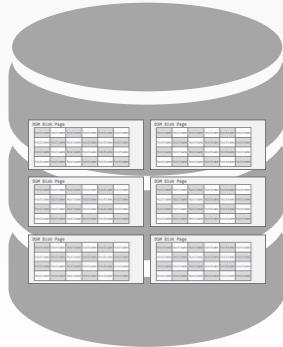
DECOMPOSITION STORAGE MODEL (DSM)

The DBMS stores the values of a single attribute for all tuples contiguously in a page.
→ Also known as a "column store".



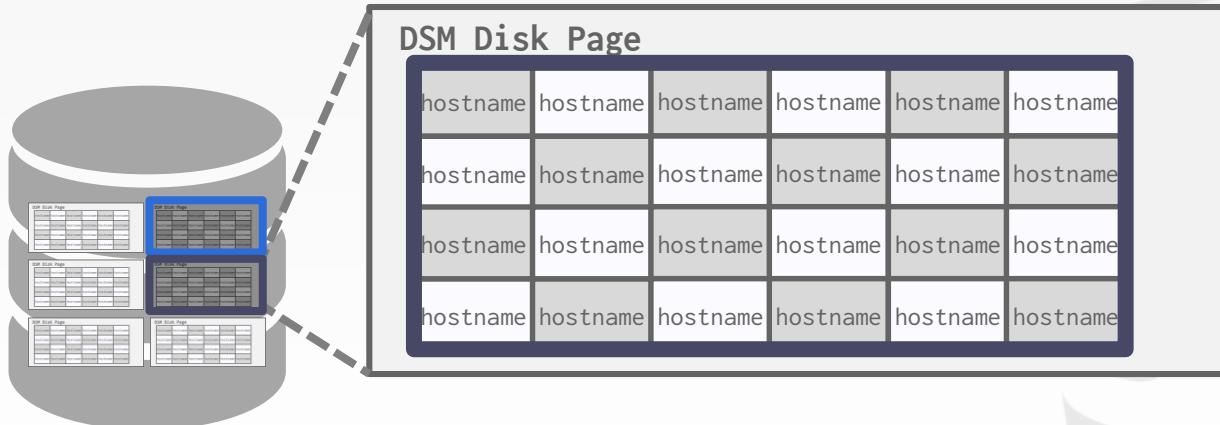
DECOMPOSITION STORAGE MODEL (DSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



DECOMPOSITION STORAGE MODEL (DSM)

```
SELECT COUNT(U.lastLogin),  
       EXTRACT(month FROM U.lastLogin) AS month  
  FROM useracct AS U  
 WHERE U.hostname LIKE '%.gov'  
 GROUP BY EXTRACT(month FROM U.lastLogin)
```



TUPLE IDENTIFICATION

Choice #1: Fixed-length Offsets

→ Each value is the same length for an attribute.

Choice #2: Embedded Tuple Ids

→ Each value is stored with its tuple id in a column.

Offsets

	A	B	C	D
0				
1				
2				
3				

Embedded Ids

	A	B	C	D
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3

DECOMPOSITION STORAGE MODEL (DSM)

Advantages

- Reduces the amount wasted I/O because the DBMS only reads the data that it needs.
- Better query processing and data compression (more on this later).

Disadvantages

- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.



DSM SYSTEM HISTORY

1970s: Cantor DBMS

1980s: DSM Proposal

1990s: SybaseIQ (in-memory only)

2000s: Vertica, VectorWise, MonetDB

2010s: Everyone

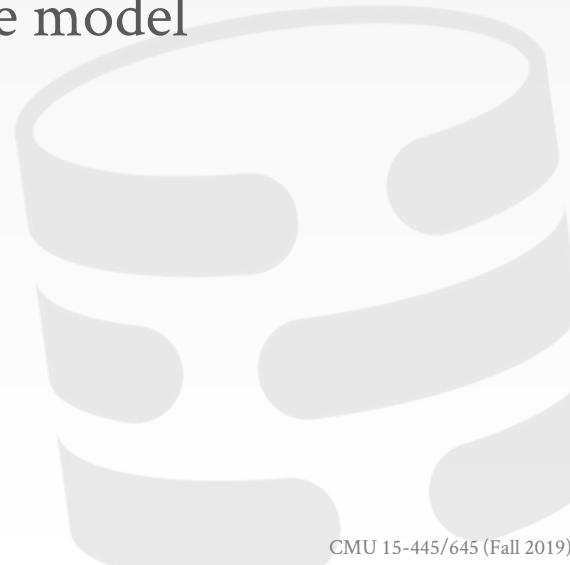


CONCLUSION

The storage manager is not entirely independent from the rest of the DBMS.

It is important to choose the right storage model for the target workload:

- OLTP = Row Store
- OLAP = Column Store



DATABASE STORAGE

Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.

← Next

05

Buffer Pools



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #1 is due TODAY @ 11:59pm

Project #1 is due Fri Sept 26th @ 11:59pm



DATABASE WORKLOADS

On-Line Transaction Processing (OLTP)

- Fast operations that only read/update a small amount of data each time.

On-Line Analytical Processing (OLAP)

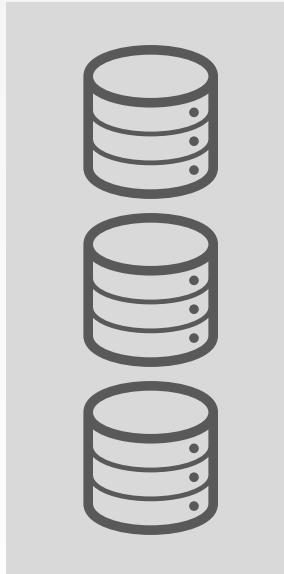
- Complex queries that read a lot of data to compute aggregates.

Hybrid Transaction + Analytical Processing

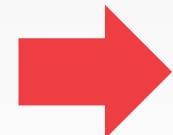
- OLTP + OLAP together on the same database instance

BIFURCATED ENVIRONMENT

 *Transactions*



*Extract
Transform
Load*



 *Analytical Queries*

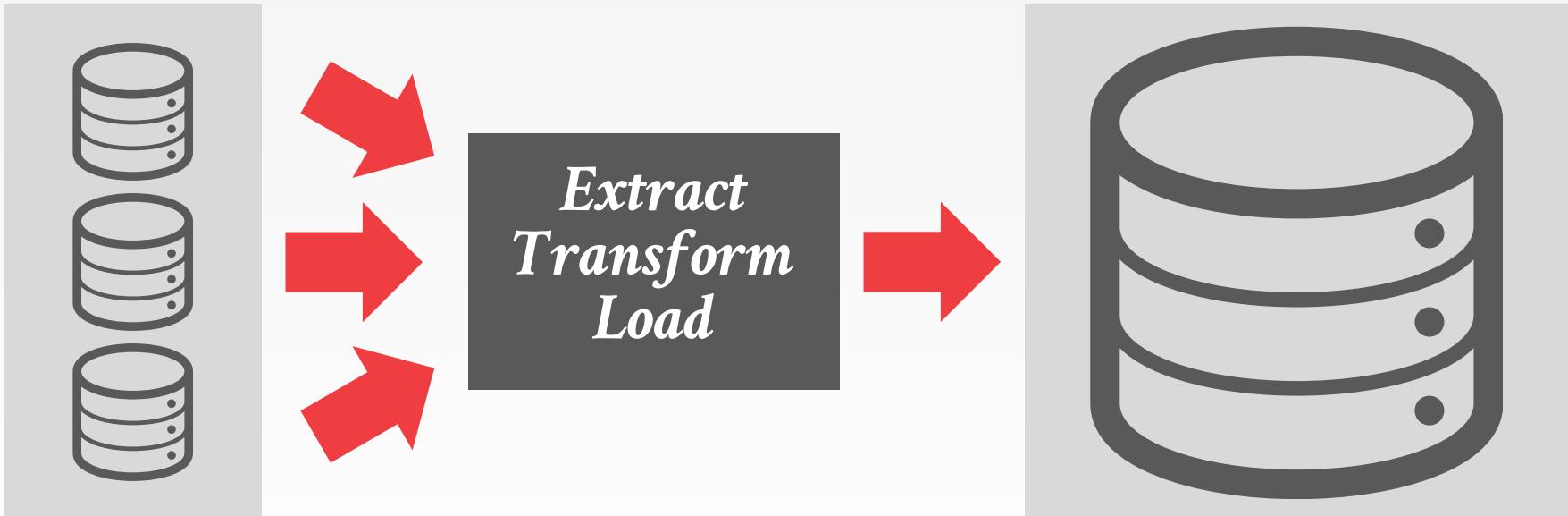


OLTP Data Silos

OLAP Data Warehouse

BIFURCATED ENVIRONMENT

 *Transactions*
 *Analytical Queries*



DATABASE STORAGE

Problem #1: How the DBMS represents the database in files on disk.

Problem #2: How the DBMS manages its memory and move data back-and-forth from disk.



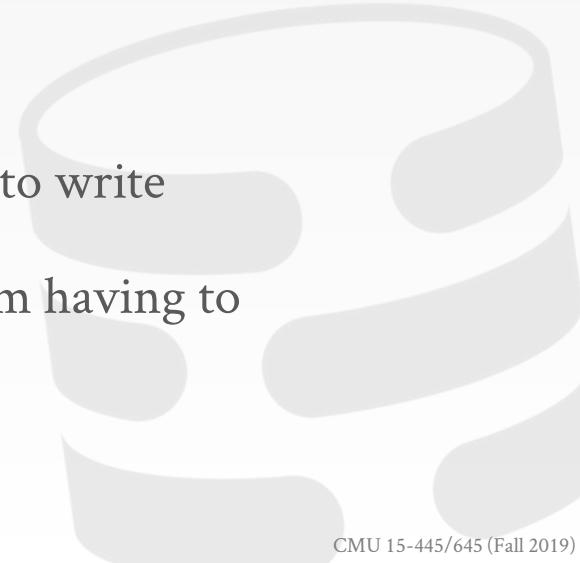
DATABASE STORAGE

Spatial Control:

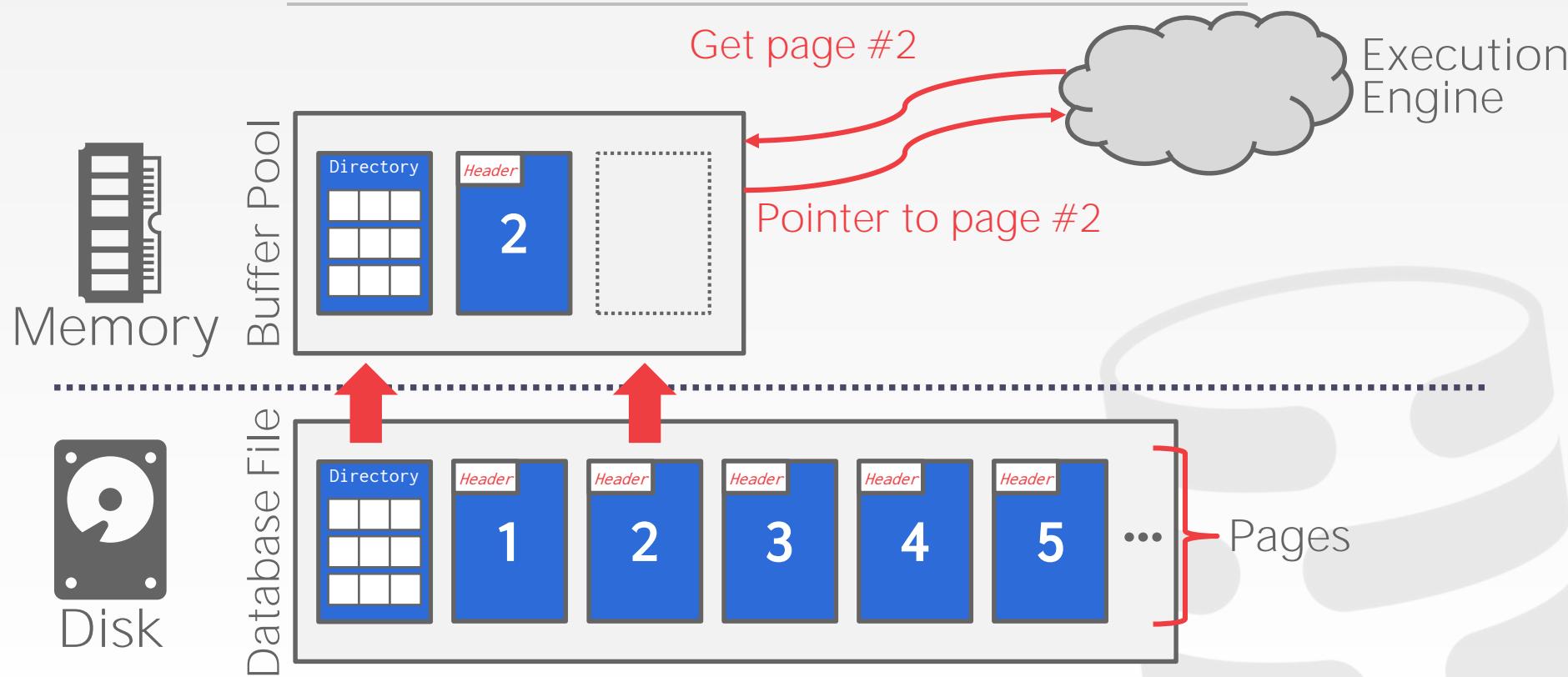
- Where to write pages on disk.
- The goal is to keep pages that are used together often as physically close together as possible on disk.

Temporal Control:

- When to read pages into memory, and when to write them to disk.
- The goal is minimize the number of stalls from having to read data from disk.



DISK-ORIENTED DBMS



TODAY'S AGENDA

Buffer Pool Manager
Replacement Policies
Other Memory Pools



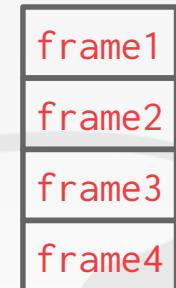
BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages.

An array entry is called a frame.

When the DBMS requests a page, an exact copy is placed into one of these frames.

Buffer Pool



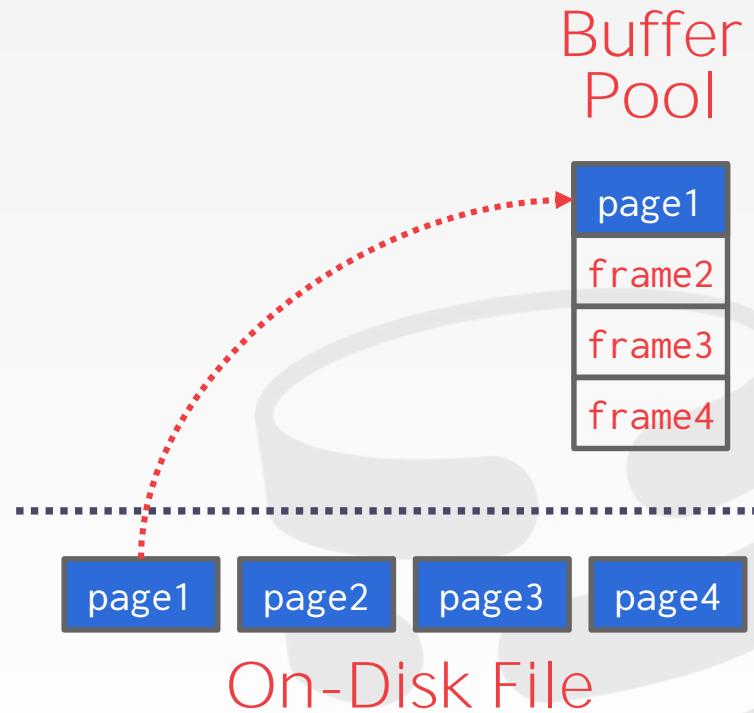
On-Disk File

BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages.

An array entry is called a frame.

When the DBMS requests a page, an exact copy is placed into one of these frames.

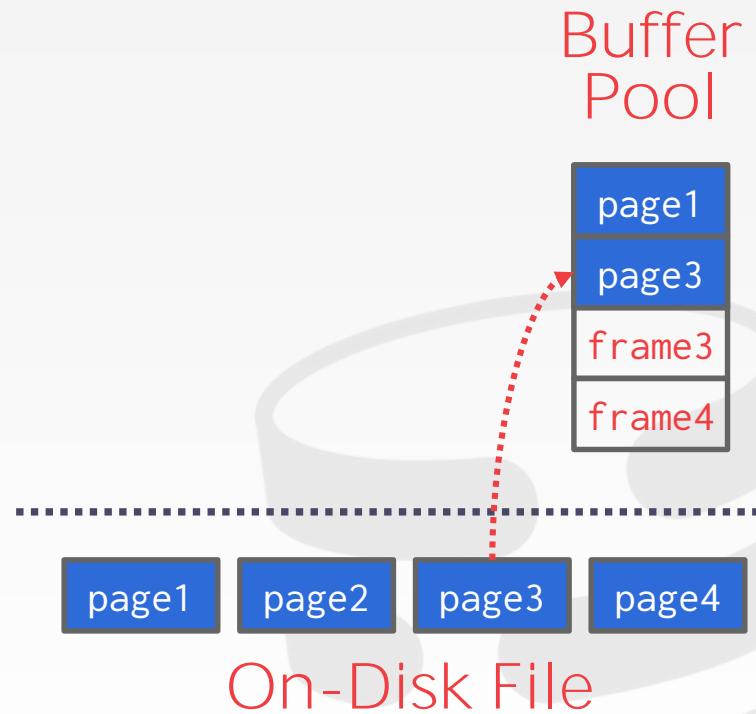


BUFFER POOL ORGANIZATION

Memory region organized as an array of fixed-size pages.

An array entry is called a frame.

When the DBMS requests a page, an exact copy is placed into one of these frames.

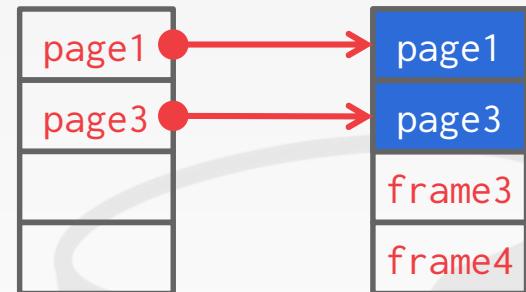


BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
→ Dirty Flag
→ Pin/Reference Counter

Page Table Buffer Pool

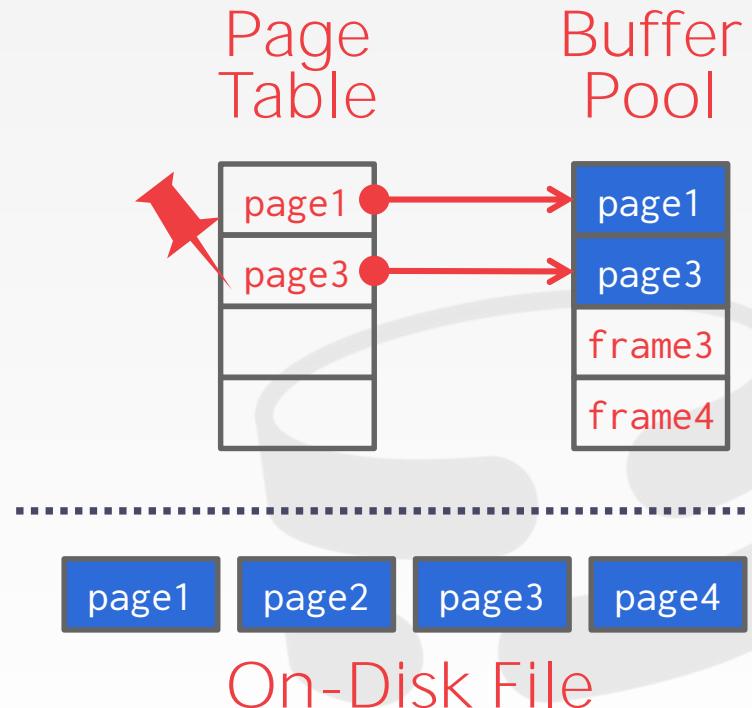


On-Disk File

BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

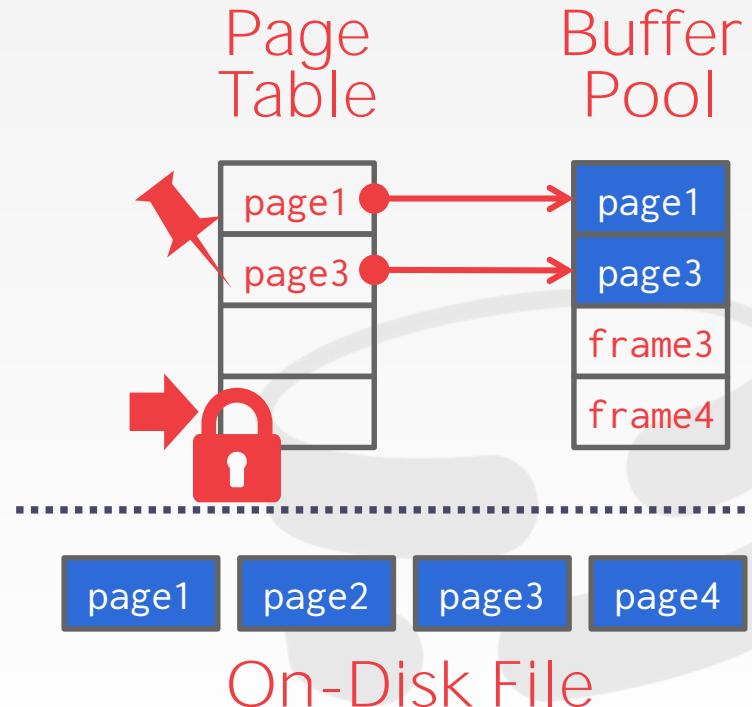
Also maintains additional meta-data per page:
→ Dirty Flag
→ Pin/Reference Counter



BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

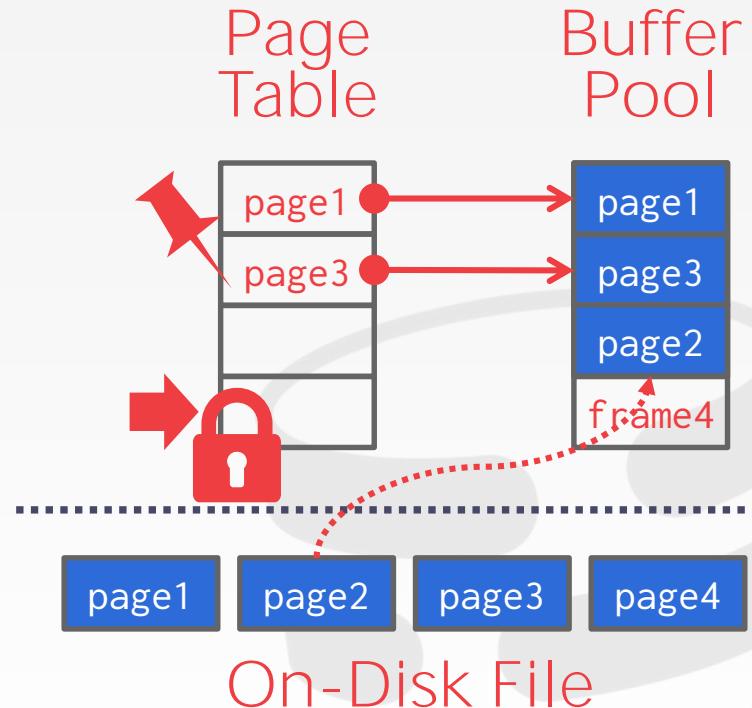
Also maintains additional meta-data per page:
→ Dirty Flag
→ Pin/Reference Counter



BUFFER POOL META-DATA

The page table keeps track of pages that are currently in memory.

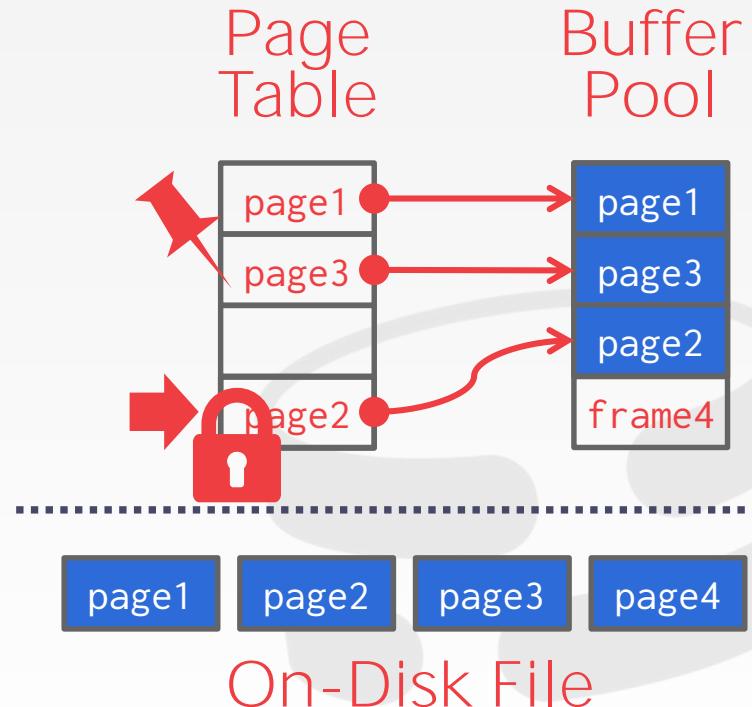
Also maintains additional meta-data per page:
→ Dirty Flag
→ Pin/Reference Counter



BUFFER POOL META-DATA

The **page table** keeps track of pages that are currently in memory.

Also maintains additional meta-data per page:
→ Dirty Flag
→ Pin/Reference Counter



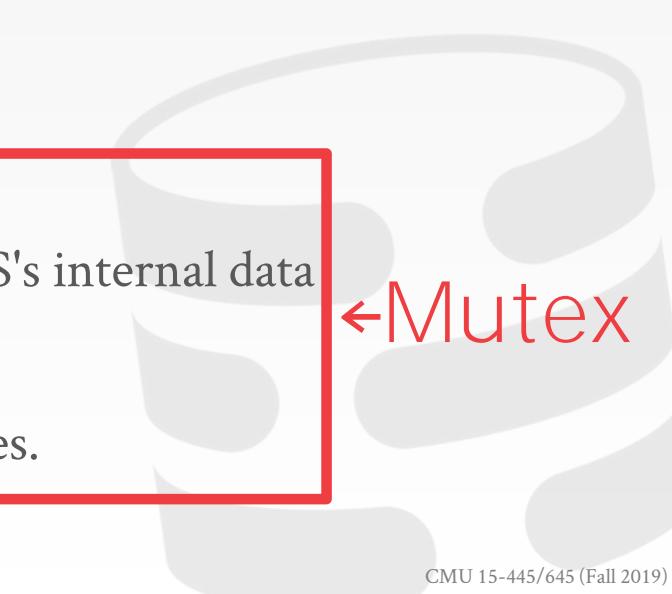
LOCKS VS. LATCHES

Locks:

- Protects the database's logical contents from other transactions.
- Held for transaction duration.
- Need to be able to rollback changes.

Latches:

- Protects the critical sections of the DBMS's internal data structure from other threads.
- Held for operation duration.
- Do not need to be able to rollback changes.



←Mutex

PAGE TABLE VS. PAGE DIRECTORY

The **page directory** is the mapping from page ids to page locations in the database files.

- All changes must be recorded on disk to allow the DBMS to find on restart.

The **page table** is the mapping from page ids to a copy of the page in buffer pool frames.

- This is an in-memory data structure that does not need to be stored on disk.

ALLOCATION POLICIES

Global Policies:

- Make decisions for all active txns.

Local Policies:

- Allocate frames to a specific txn without considering the behavior of concurrent txns.
- Still need to support sharing pages.



BUFFER POOL OPTIMIZATIONS

Multiple Buffer Pools

Pre-Fetching

Scan Sharing

Buffer Pool Bypass



MULTIPLE BUFFER POOLS

The DBMS does not always have a single buffer pool for the entire system.

- Multiple buffer pool instances
- Per-database buffer pool
- Per-page type buffer pool

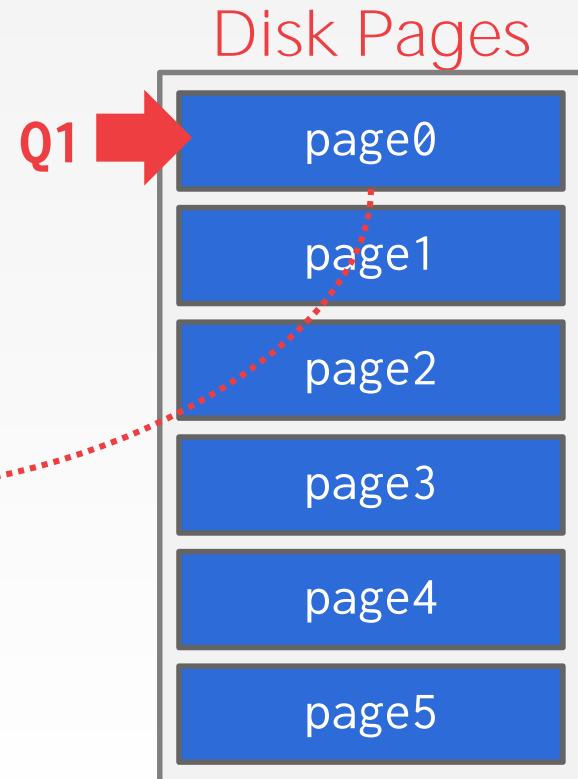
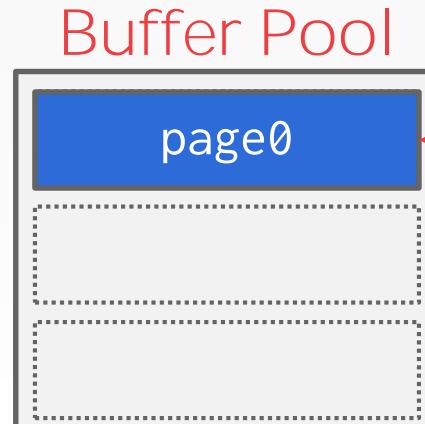
Helps reduce latch contention and improve locality.



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans



PRE-FETCHING

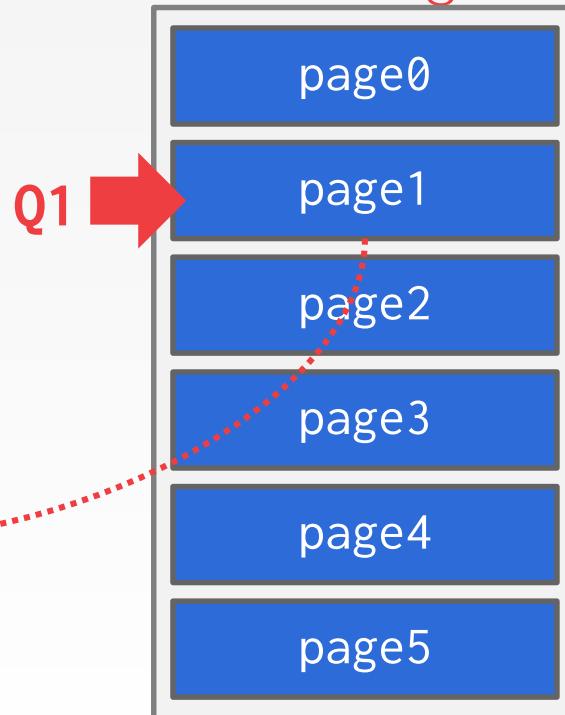
The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans

Buffer Pool



Disk Pages



PRE-FETCHING

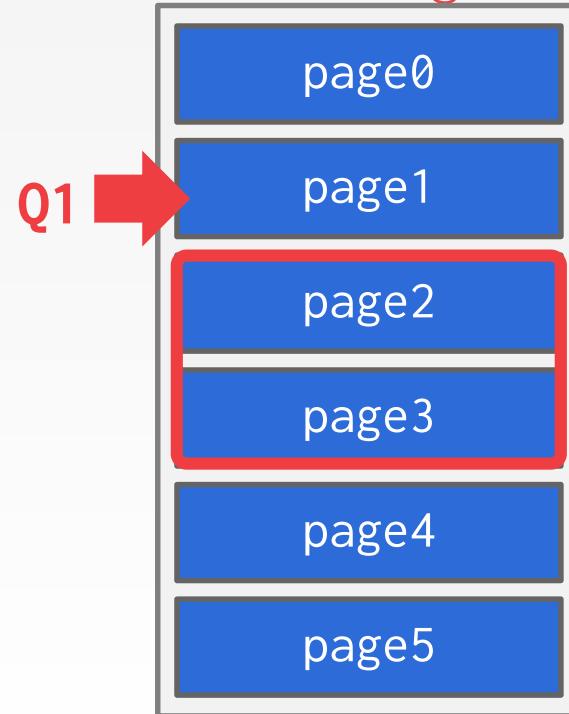
The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans

Buffer Pool



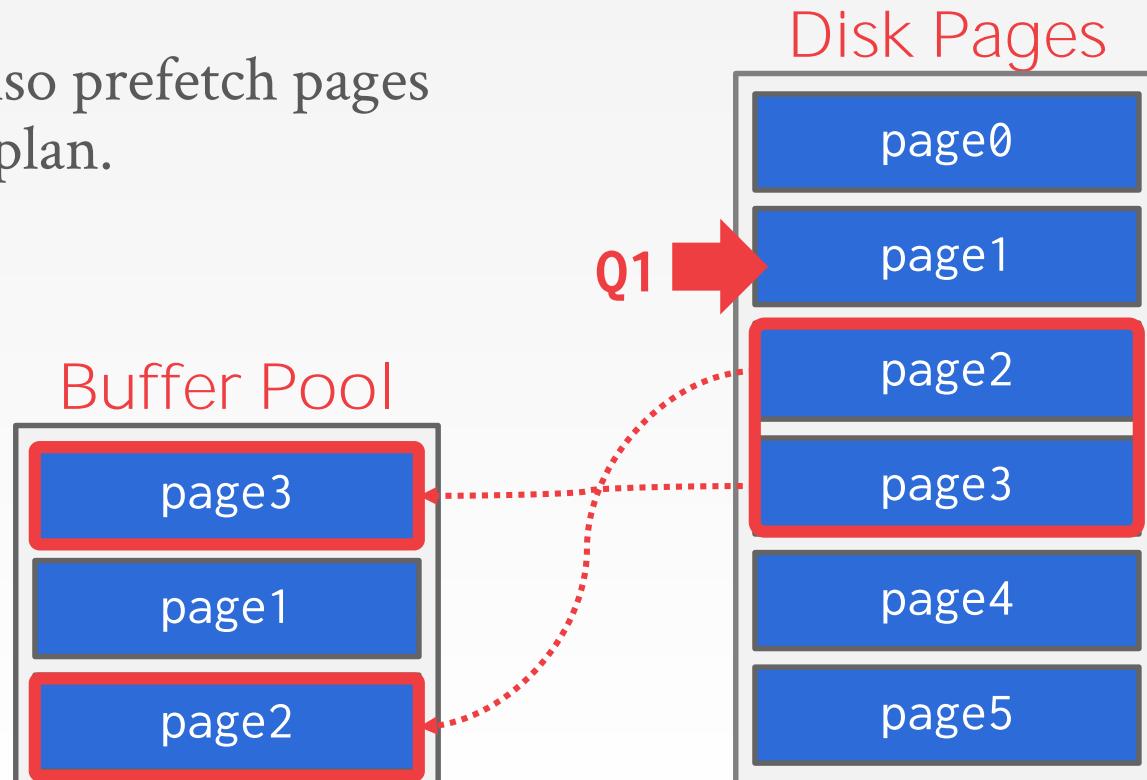
Disk Pages



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans



PRE-FETCHING

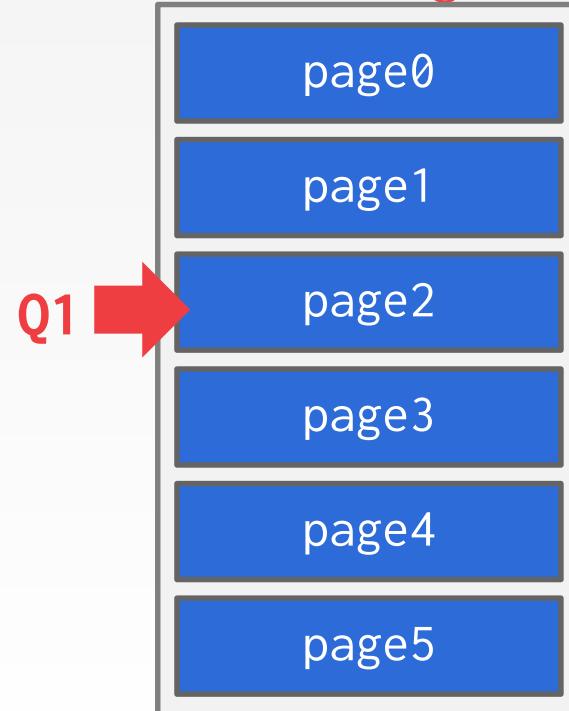
The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans

Buffer Pool



Disk Pages



PRE-FETCHING

The DBMS can also prefetch pages based on a query plan.

- Sequential Scans
- Index Scans

Buffer Pool



Disk Pages



PRE-FETCHING

Q1

```
SELECT * FROM A  
WHERE val BETWEEN 100 AND 250
```

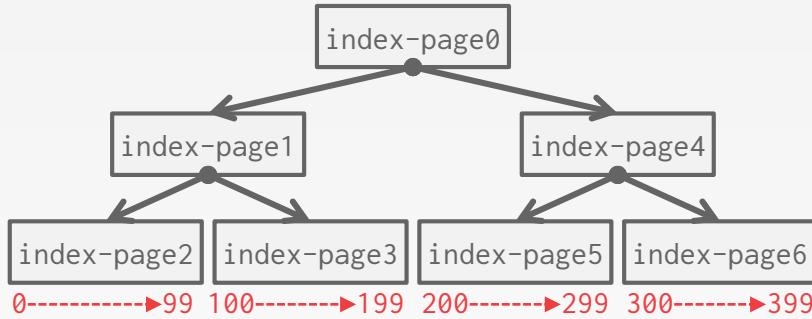
Buffer Pool



Disk Pages



PRE-FETCHING



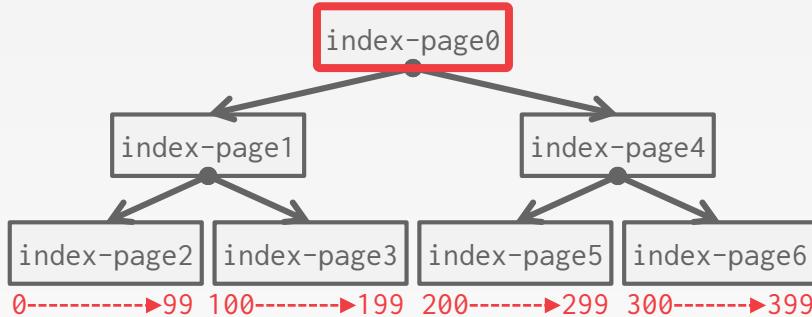
Buffer Pool



Disk Pages



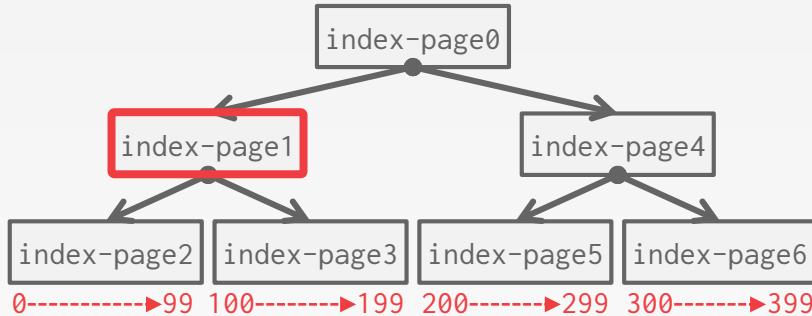
PRE-FETCHING



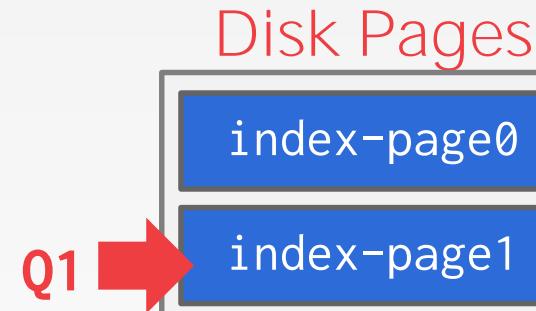
Buffer Pool



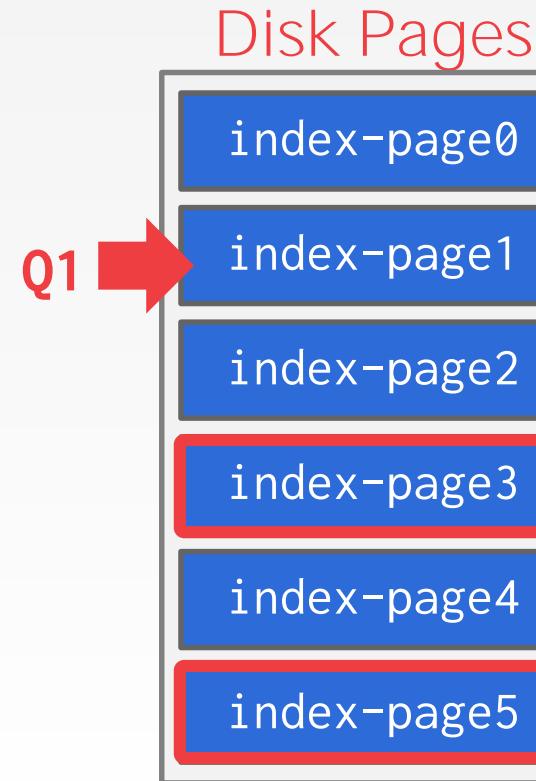
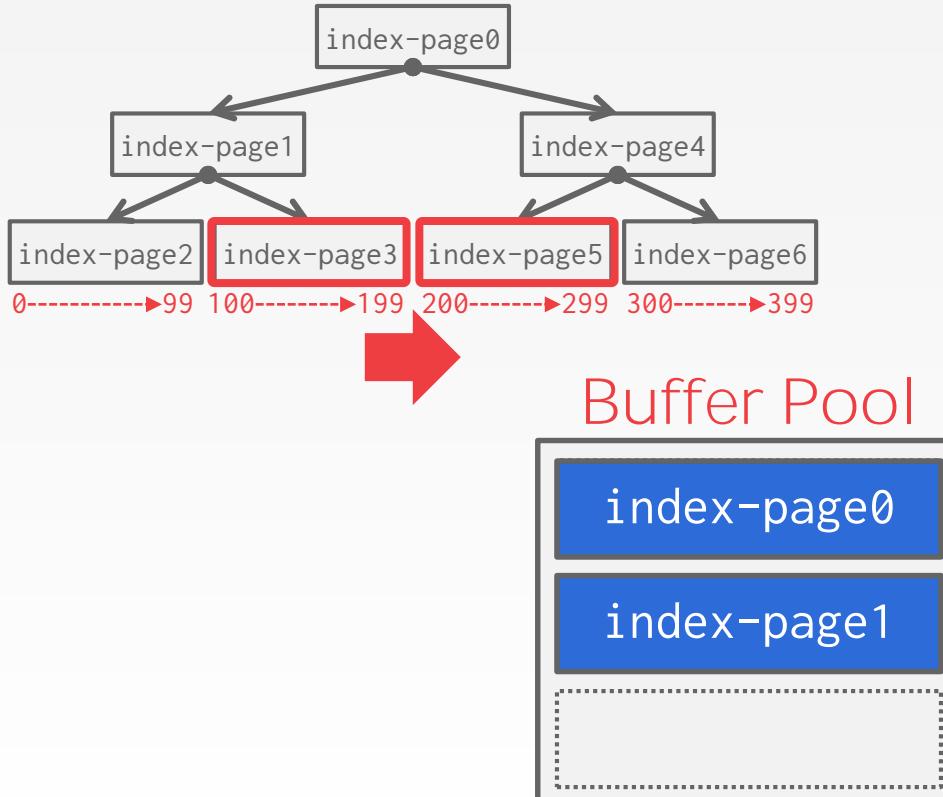
PRE-FETCHING



Buffer Pool



PRE-FETCHING



SCAN SHARING

Queries can reuse data retrieved from storage or operator computations.

- This is different from result caching.

Allow multiple queries to attach to a single cursor that scans a table.

- Queries do not have to be the same.
- Can also share intermediate results.



SCAN SHARING

If a query starts a scan and if there one already doing this, then the DBMS will attach to the second query's cursor.

→ The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure.

Fully supported in IBM DB2 and MSSQL.

Oracle only supports cursor sharing for identical queries.



Microsoft®
SQL Server

IBM DB2

ORACLE

SCAN SHARING

Q1

SELECT SUM(val) FROM A

Buffer Pool



Disk Pages

Q1



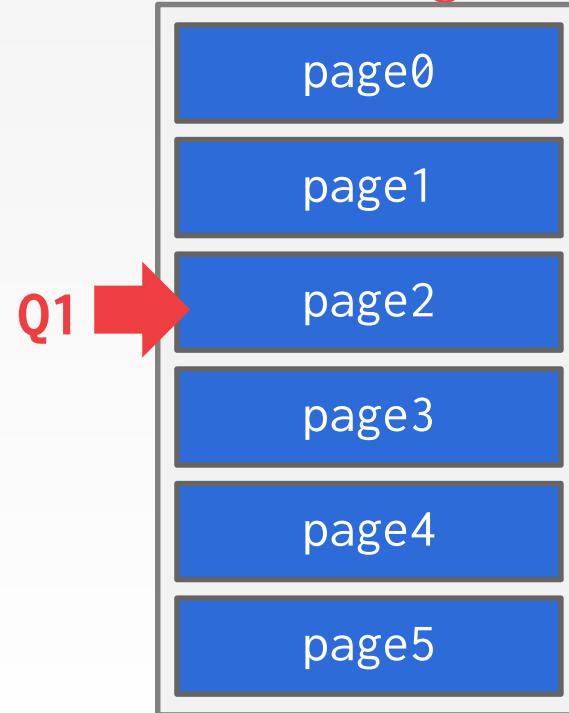
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Buffer Pool



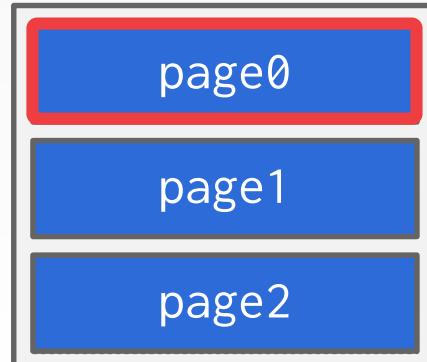
Disk Pages



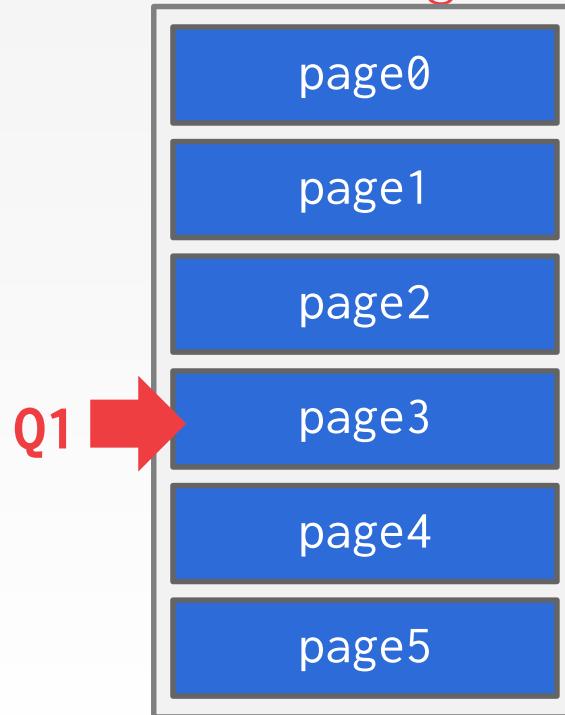
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Buffer Pool



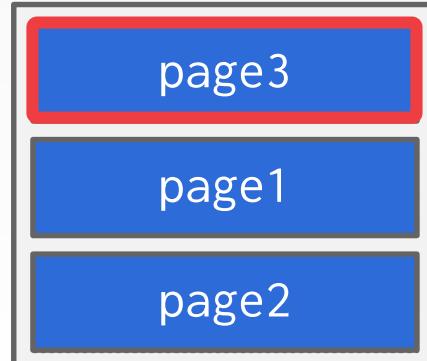
Disk Pages



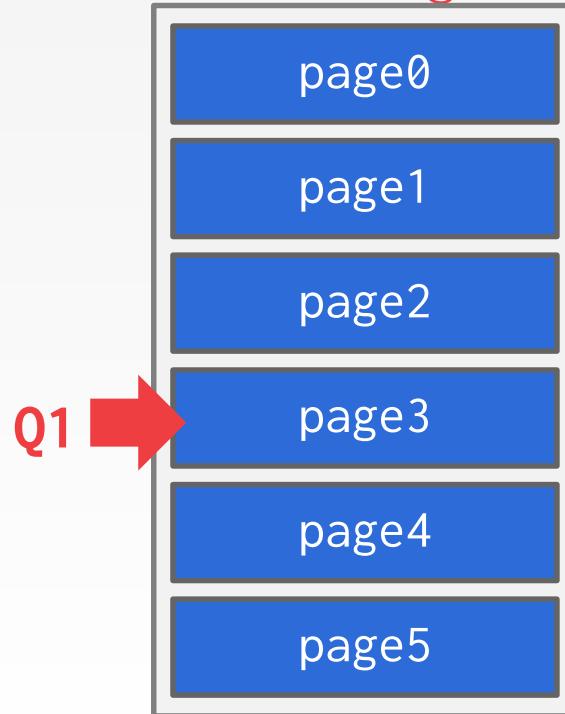
SCAN SHARING

Q1 `SELECT SUM(val) FROM A`

Buffer Pool



Disk Pages



SCAN SHARING

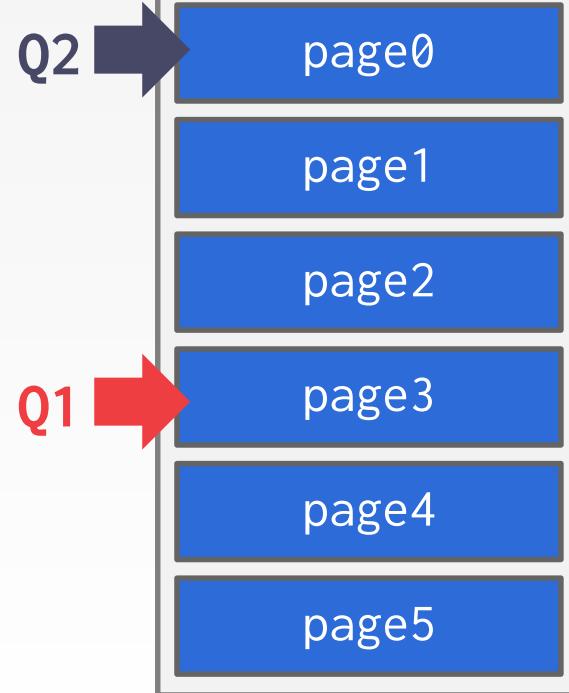
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



SCAN SHARING

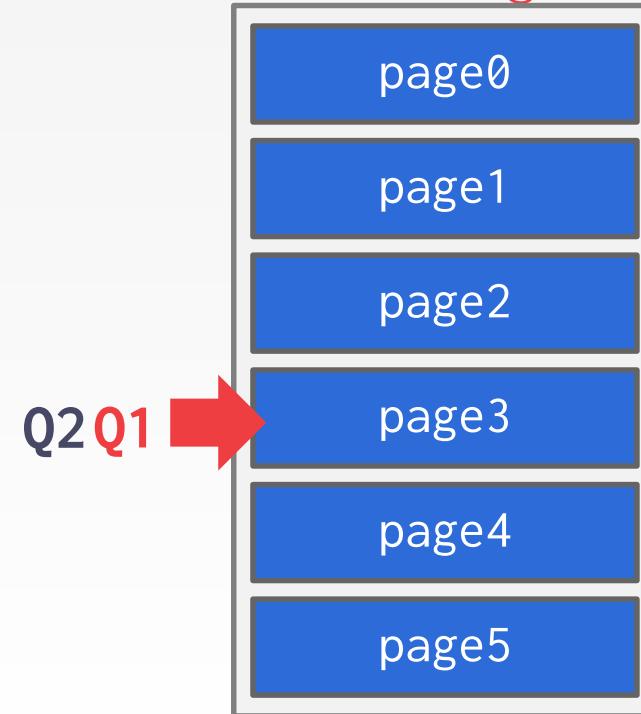
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



SCAN SHARING

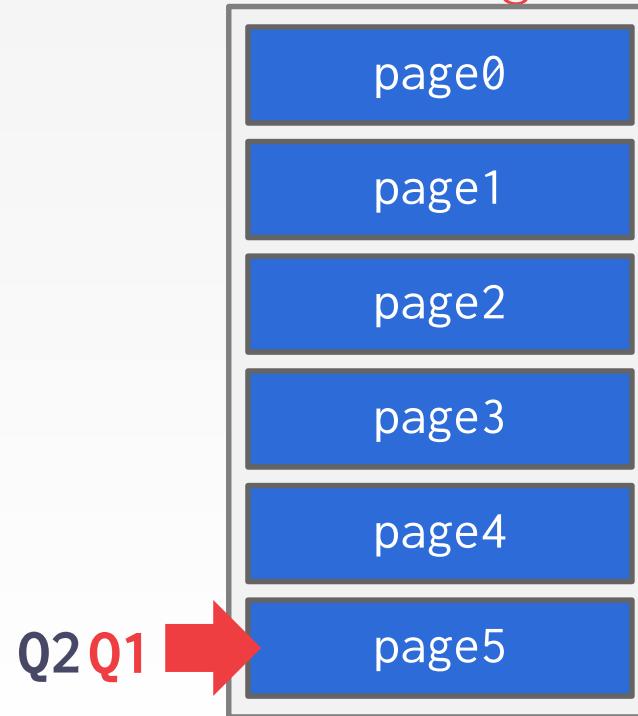
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



SCAN SHARING

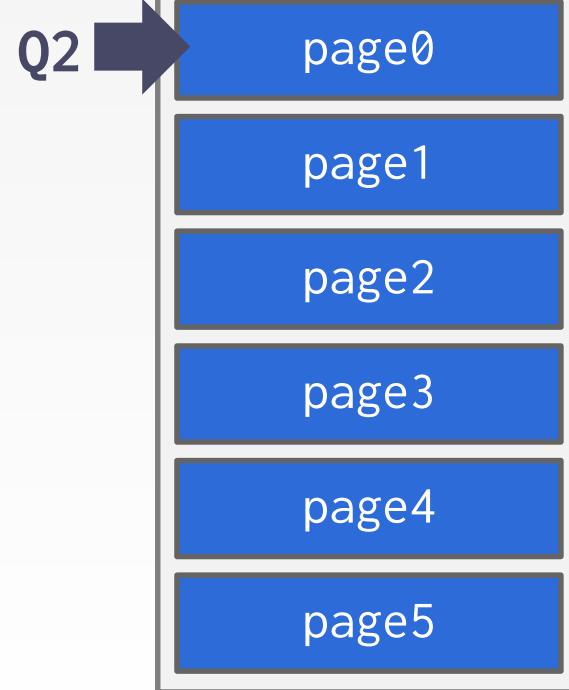
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



SCAN SHARING

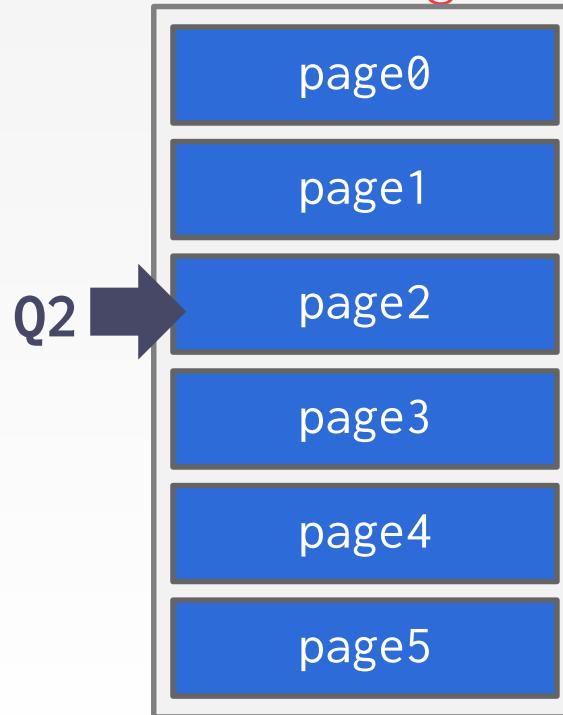
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



SCAN SHARING

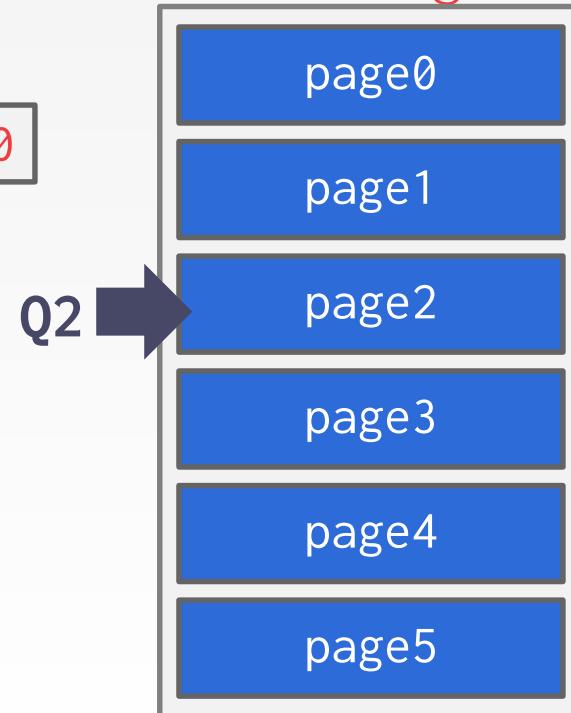
Q1 `SELECT SUM(val) FROM A`

Q2 `SELECT AVG(val) FROM A LIMIT 100`

Buffer Pool



Disk Pages



BUFFER POOL BYPASS

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.

- Memory is local to running query.
- Works well if operator needs to read a large sequence of pages that are contiguous on disk.
- Can also be used for temporary data (sorting, joins).

Called "Light Scans" in Informix.

ORACLE

Microsoft®
SQL Server

PostgreSQL

Informix®

OS PAGE CACHE

Most disk operations go through the OS API.

Unless you tell it not to, the OS maintains its own filesystem cache.

Most DBMSs use direct I/O (O_DIRECT) to bypass the OS's cache.

- Redundant copies of pages.
- Different eviction policies.

Demo: Postgres

BUFFER REPLACEMENT POLICIES

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.

Goals:

- Correctness
- Accuracy
- Speed
- Meta-data overhead



LEAST-RECENTLY USED

Maintain a timestamp of when each page was last accessed.

When the DBMS needs to evict a page, select the one with the oldest timestamp.

→ Keep the pages in sorted order to reduce the search time on eviction.

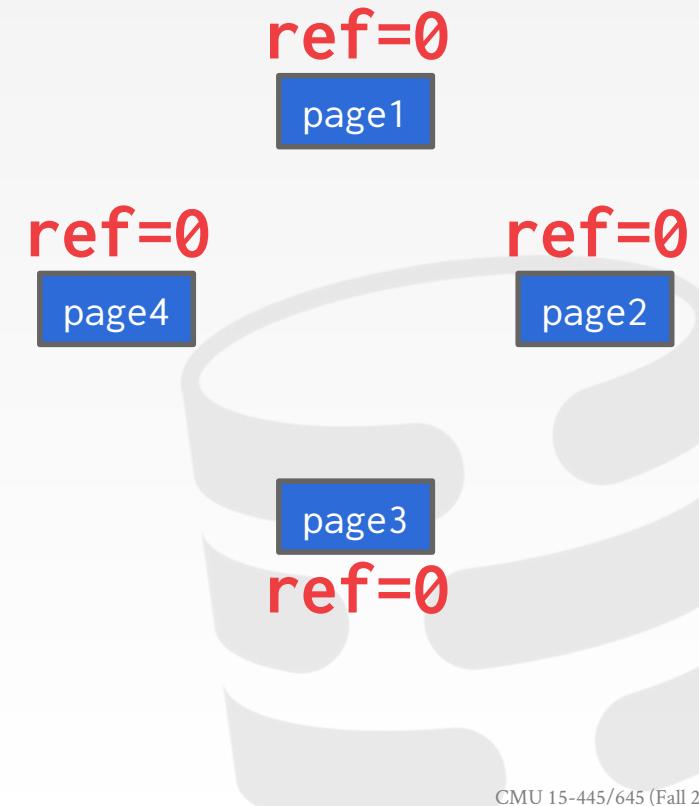
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



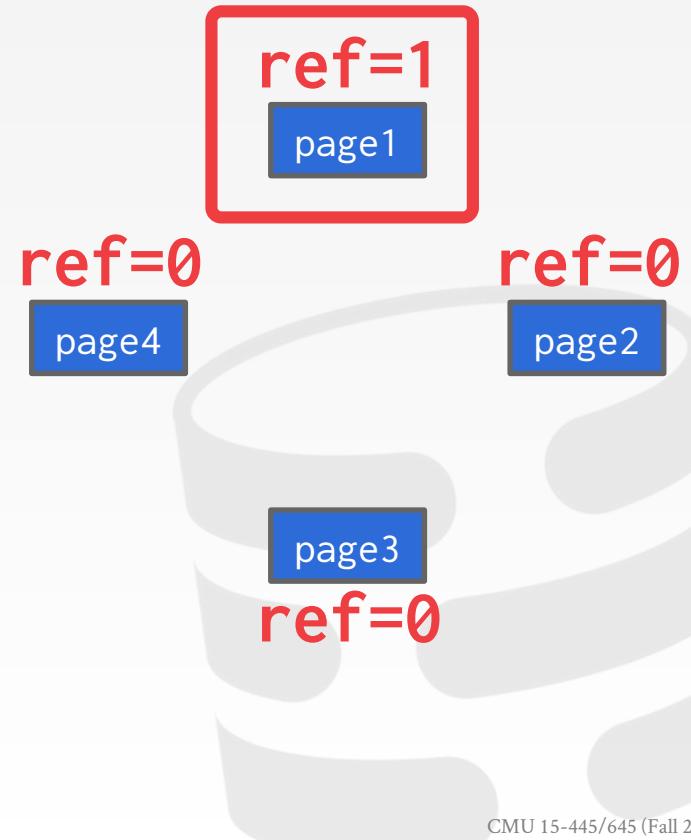
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



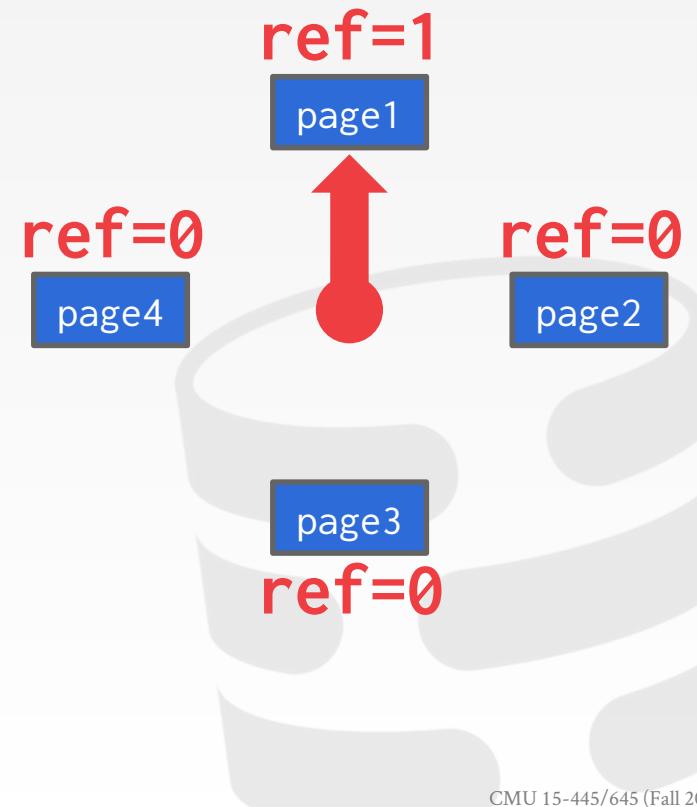
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



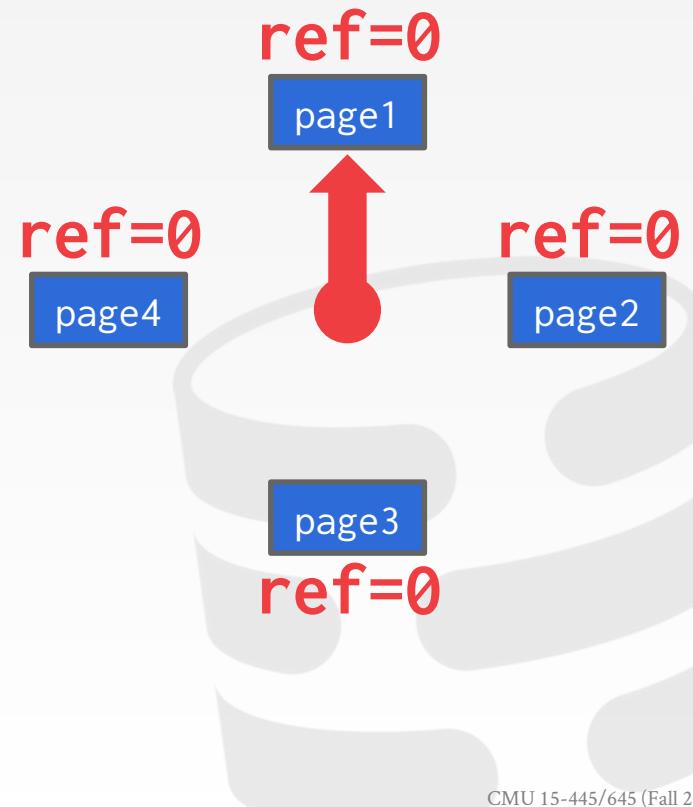
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



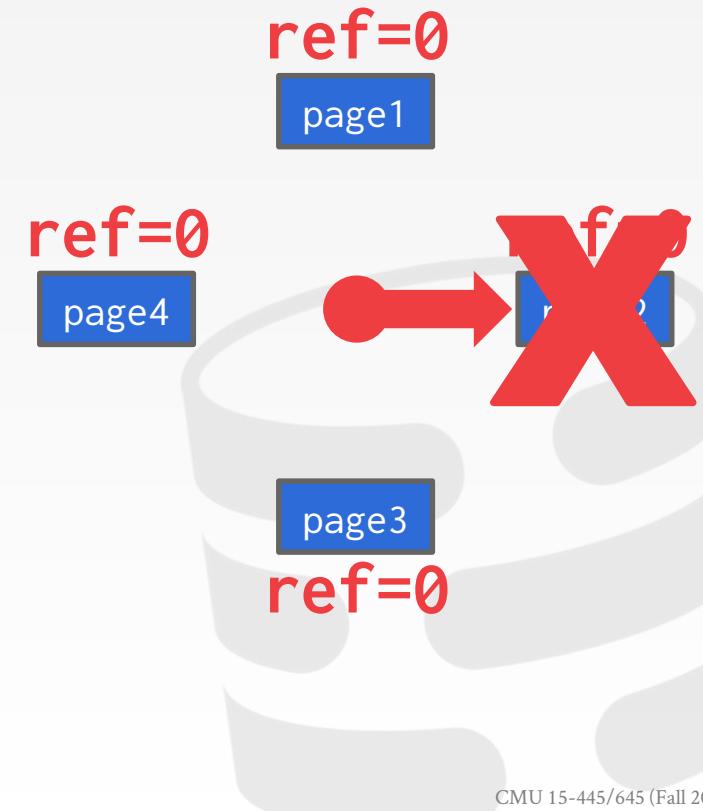
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



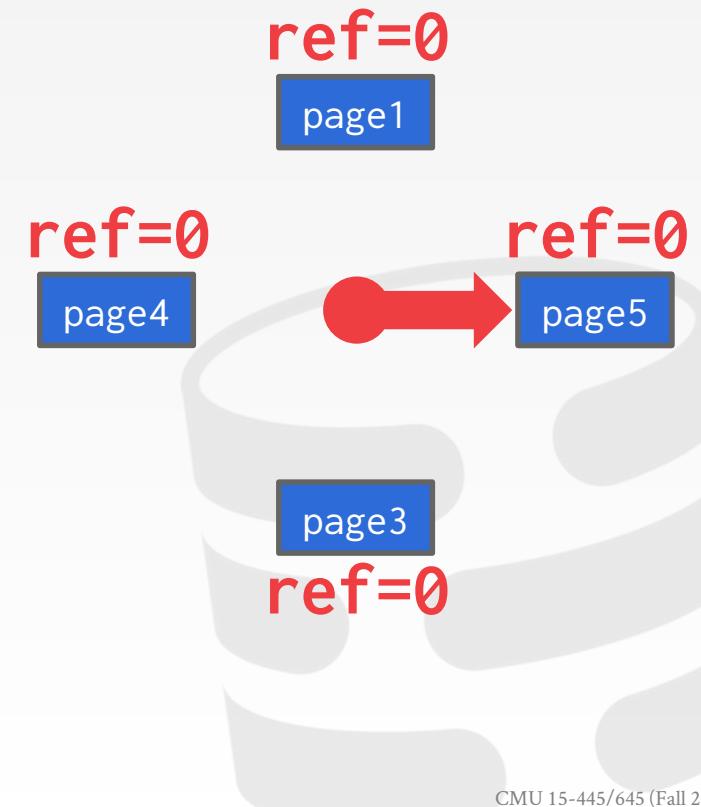
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



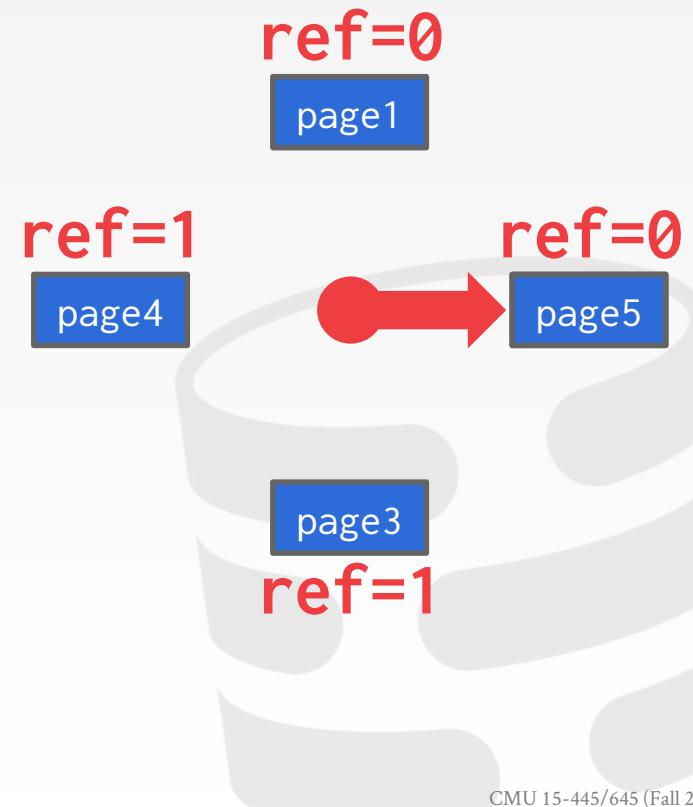
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.



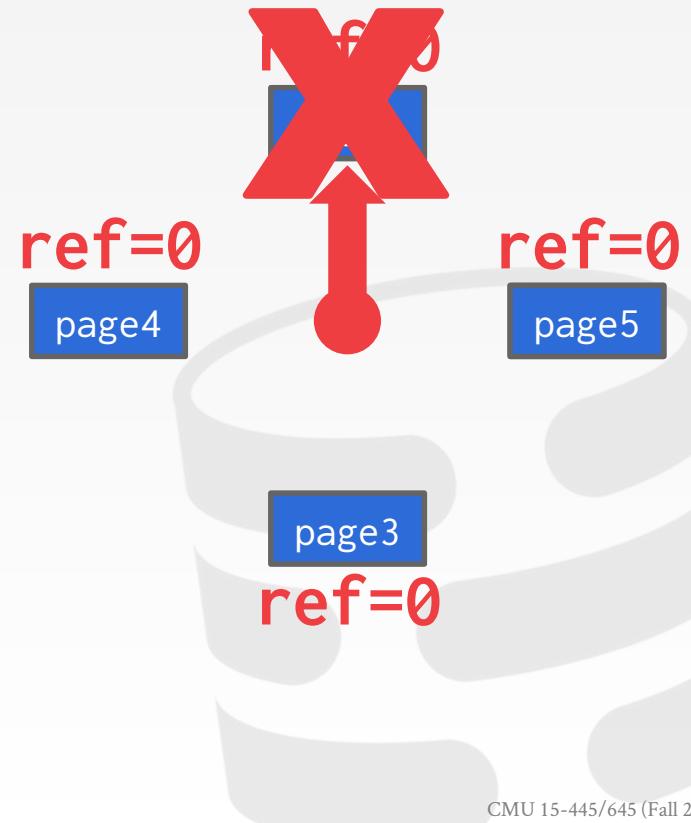
CLOCK

Approximation of LRU without needing a separate timestamp per page.

- Each page has a reference bit.
- When a page is accessed, set to 1.

Organize the pages in a circular buffer with a "clock hand":

- Upon sweeping, check if a page's bit is set to 1.
- If yes, set to zero. If no, then evict.

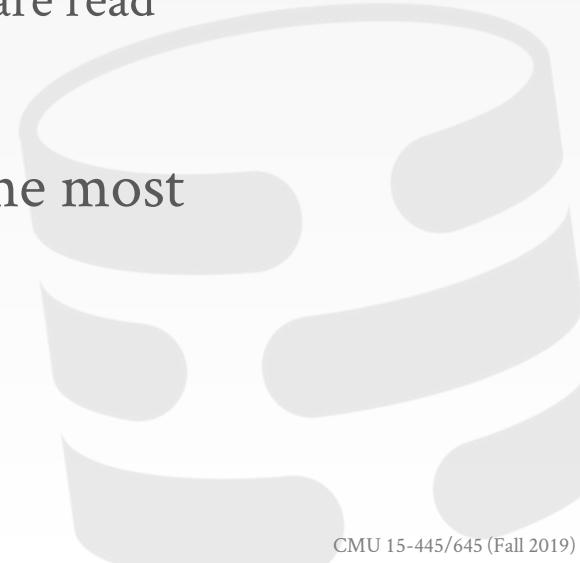


PROBLEMS

LRU and CLOCK replacement policies are susceptible to sequential flooding.

- A query performs a sequential scan that reads every page.
- This pollutes the buffer pool with pages that are read once and then never again.

The most recently used page is actually the most unneeded page.



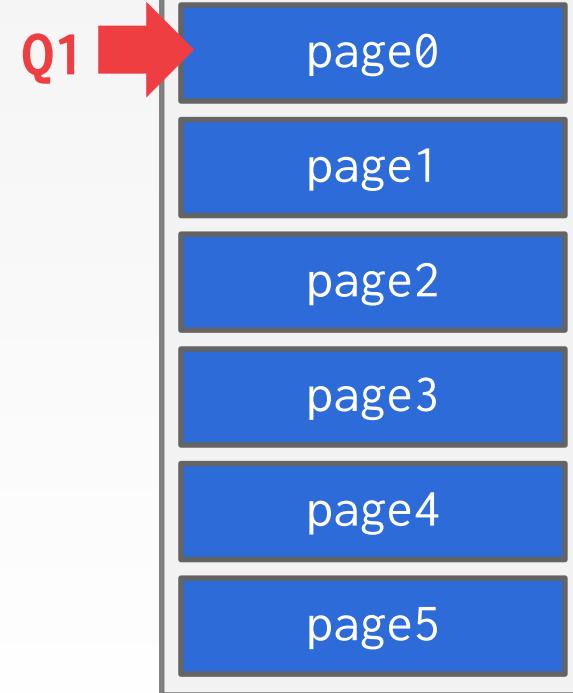
SEQUENTIAL FLOODING

Q1 **SELECT * FROM A WHERE id = 1**

Buffer Pool



Disk Pages



SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages

Q2

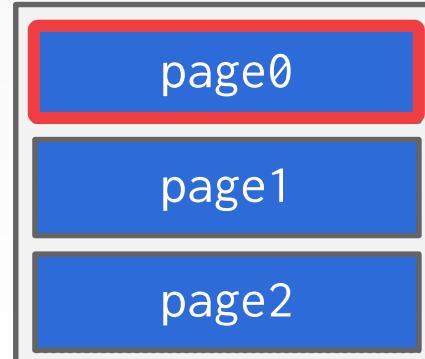


SEQUENTIAL FLOODING

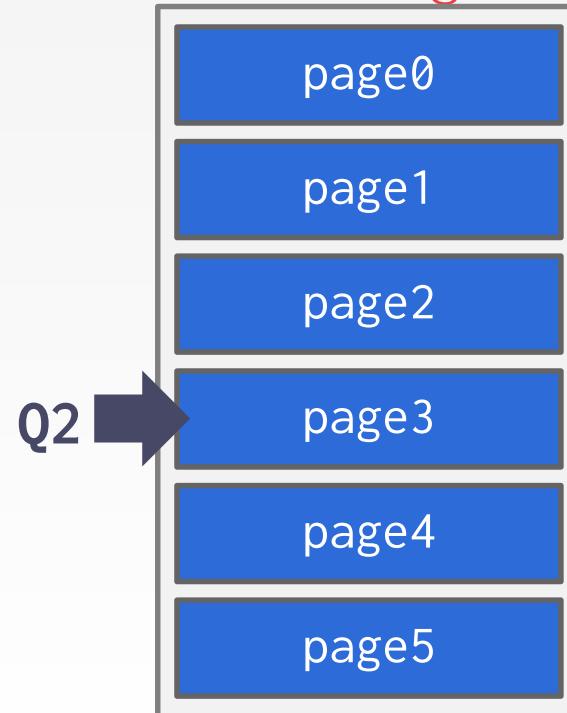
Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



SEQUENTIAL FLOODING

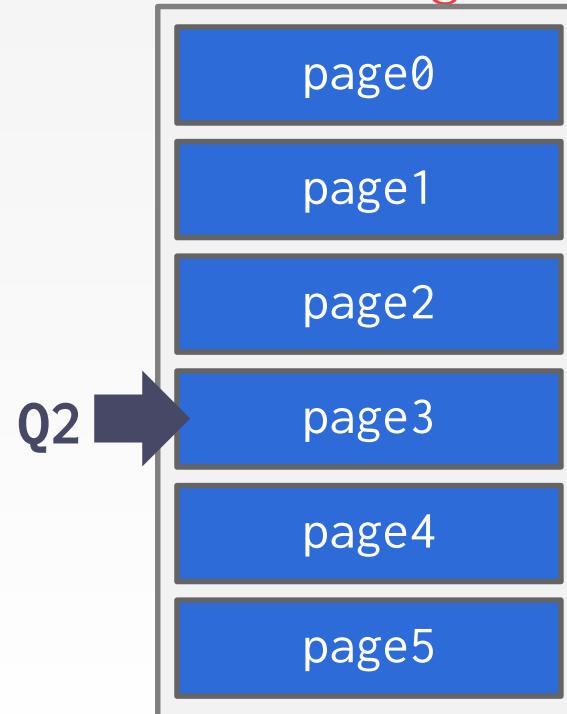
Q1 `SELECT * FROM A WHERE id = 1`

Q2 `SELECT AVG(val) FROM A`

Buffer Pool



Disk Pages



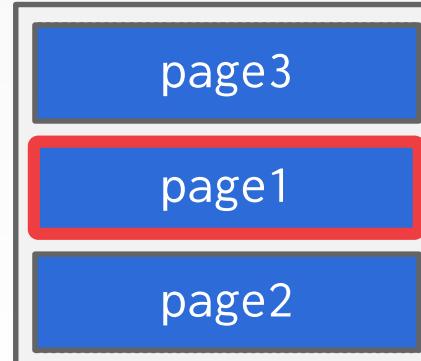
SEQUENTIAL FLOODING

Q1 `SELECT * FROM A WHERE id = 1`

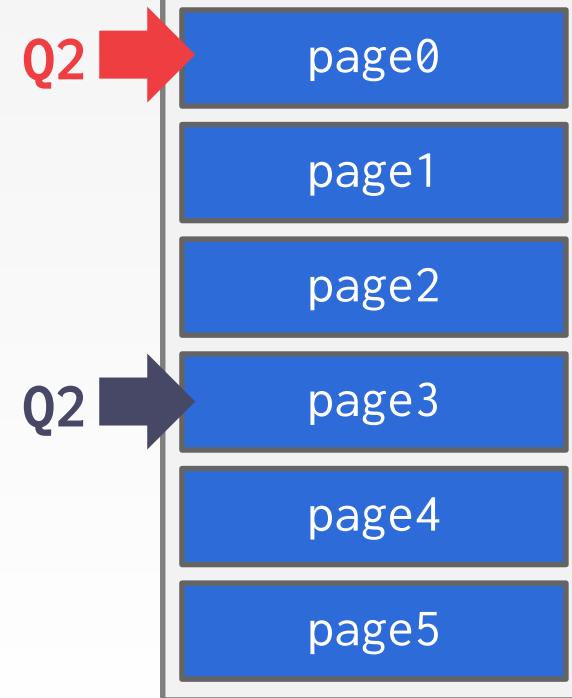
Q2 `SELECT AVG(val) FROM A`

Q3 `SELECT * FROM A WHERE id = 1`

Buffer Pool



Disk Pages



BETTER POLICIES: LRU-K

Track the history of the last K references as timestamps and compute the interval between subsequent accesses.

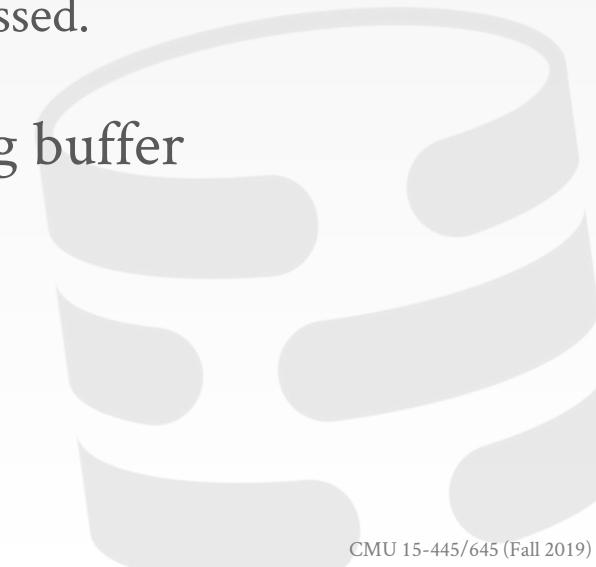
The DBMS then uses this history to estimate the next time that page is going to be accessed.

BETTER POLICIES: LOCALIZATION

The DBMS chooses which pages to evict on a per txn/query basis. This minimizes the pollution of the buffer pool from each query.

→ Keep track of the pages that a query has accessed.

Example: Postgres maintains a small ring buffer that is private to the query.

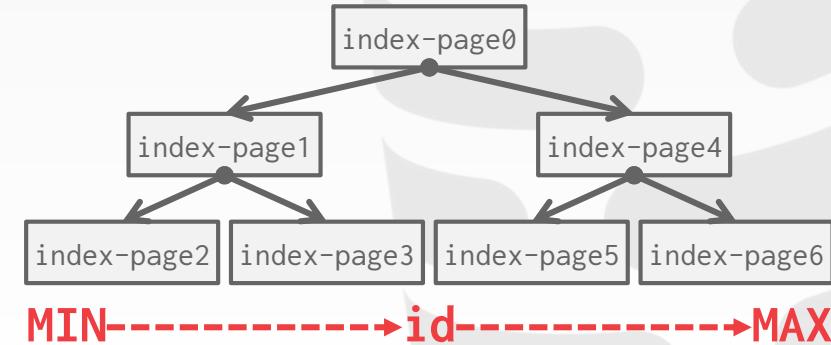


BETTER POLICIES: PRIORITY HINTS

The DBMS knows what the context of each page during query execution.

It can provide hints to the buffer pool on whether a page is important or not.

Q1 `INSERT INTO A VALUES (id++)`

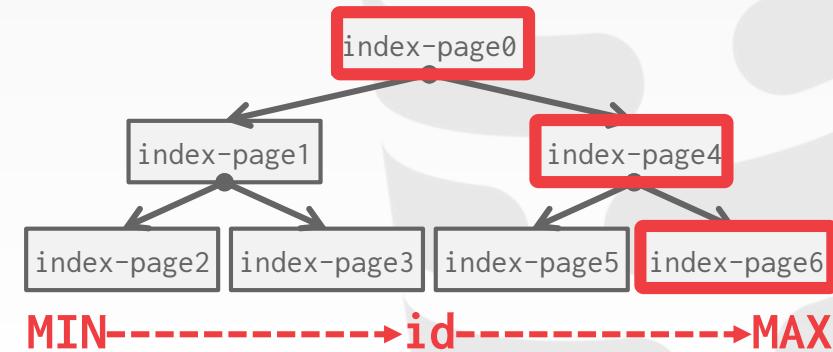


BETTER POLICIES: PRIORITY HINTS

The DBMS knows what the context of each page during query execution.

It can provide hints to the buffer pool on whether a page is important or not.

Q1 `INSERT INTO A VALUES (id++)`



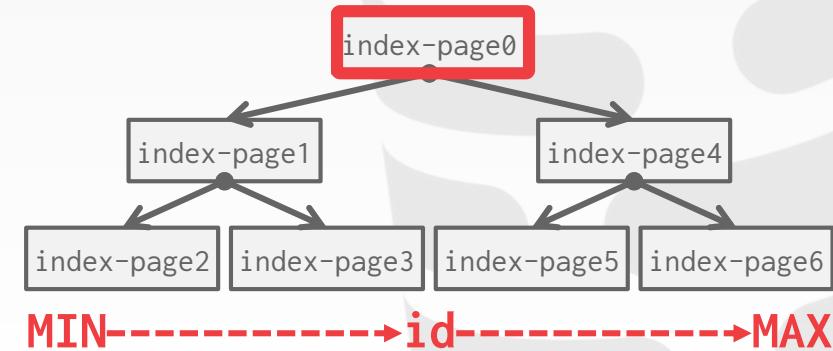
BETTER POLICIES: PRIORITY HINTS

The DBMS knows what the context of each page during query execution.

It can provide hints to the buffer pool on whether a page is important or not.

Q1 `INSERT INTO A VALUES (id++)`

Q2 `SELECT * FROM A WHERE id = ?`



DIRTY PAGES

FAST: If a page in the buffer pool is not dirty, then the DBMS can simply "drop" it.

SLOW: If a page is dirty, then the DBMS must write back to disk to ensure that its changes are persisted.

Trade-off between fast evictions versus dirty writing pages that will not be read again in the future.

BACKGROUND WRITING

The DBMS can periodically walk through the page table and write dirty pages to disk.

When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.

Need to be careful that we don't write dirty pages before their log records have been written...

OTHER MEMORY POOLS

The DBMS needs memory for things other than just tuples and indexes.

These other memory pools may not always be backed by disk. Depends on implementation.

- Sorting + Join Buffers
- Query Caches
- Maintenance Buffers
- Log Buffers
- Dictionary Caches



CONCLUSION

The DBMS can manage that sweet, sweet memory better than the OS.

Leverage the semantics about the query plan to make better decisions:

- Evictions
- Allocations
- Pre-fetching

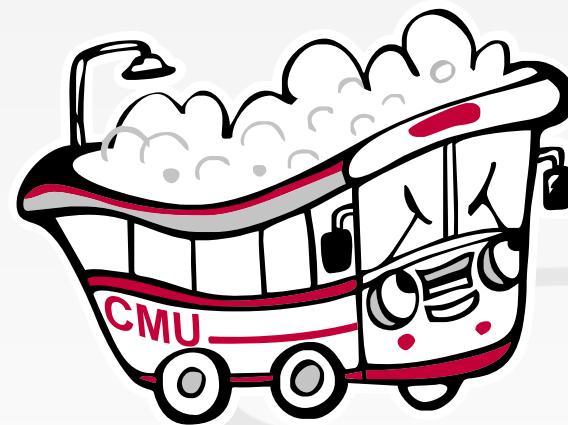


PROJECT #1

You will build the first component of your storage manager.

- Clock Replacement Policy
- Buffer Pool Manager

We will provide you with the disk manager and page layouts.



BusTub

Due Date:
Friday Sept 27th @ 11:59pm

TASK #1 – CLOCK REPLACEMENT POLICY

Build a data structure that tracks the usage of **frame_ids** using the CLOCK policy.

General Hints:

- Your **ClockReplacer** needs to check the "pinned" status of a **Page**.
- If there are no pages touched since last sweep, then return the lowest page id.

TASK #2 – BUFFER POOL MANAGER

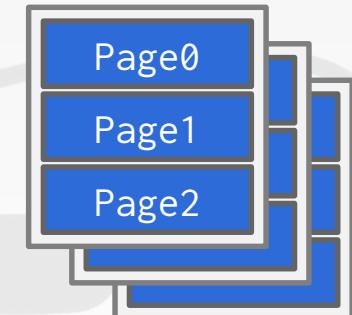
Use your CLOCK replacer to manage the allocation of pages.

- Need to maintain an internal data structures of allocated + free pages.
- We will provide you components to read/write data from disk.
- Use whatever data structure you want for the page table.

Buffer Pool
(In-Memory)



Database
(On-Disk)



General Hints:

- Make sure you get the order of operations correct when pinning.

GETTING STARTED

Download the source code from [GitHub](#).

Make sure you can build it on your machine.

- We've tested Ubuntu, OSX, and Windows (WSL2).
- We are also providing a docker file to setup your environment.
- It does not compile on the Andrews machines. Please contact me if this is a problem.

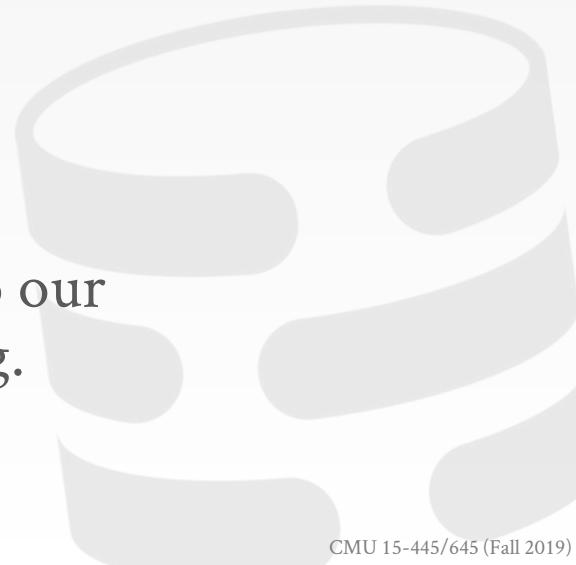
THINGS TO NOTE

Do not change any file other than the four that you must hand in.

The projects are cumulative.

We will not be providing solutions.

Post your questions on Piazza or come to our office hours. We will not help you debug.



CODE QUALITY

We will automatically check whether you are writing good code.

- [Google C++ Style Guide](#)
- [Doxxygen Javadoc Style](#)

You need to run these targets before you submit your implementation to Gradescope.

- [**make format**](#)
- [**make check-lint**](#)
- [**make check-censored**](#)
- [**make check-clang-tidy**](#)



PLAGIARISM WARNING

Your project implementation must be your own work.

- You may not copy source code from other groups or the web.
- Do not publish your implementation on GitHub.

Plagiarism will not be tolerated.

See [CMU's Policy on Academic Integrity](#) for additional information.



NEXT CLASS

HASH TABLES!



06

Hash Tables



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 is due Fri Sept 27th @ 11:59pm

Homework #2 is due Mon Sept 30th @ 11:59pm



COURSE STATUS

We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.

Two types of data structures:
→ Hash Tables
→ Trees

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

DATA STRUCTURES

Internal Meta-data

Core Data Storage

Temporary Data Structures

Table Indexes



DESIGN DECISIONS

Data Organization

→ How we layout data structure in memory/pages and what information to store to support efficient access.

Concurrency

→ How to enable multiple threads to access the data structure at the same time without causing problems.

HASH TABLES

A hash table implements an unordered associative array that maps keys to values.

It uses a hash function to compute an offset into the array for a given key, from which the desired value can be found.

Space Complexity: **O(n)**

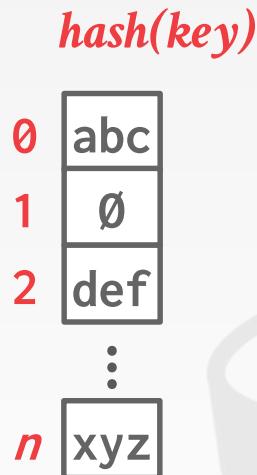
Operation Complexity:

- Average: **O(1)** ← *Money cares about constants!*
- Worst: **O(n)**

STATIC HASH TABLE

Allocate a giant array that has one slot for every element you need to store.

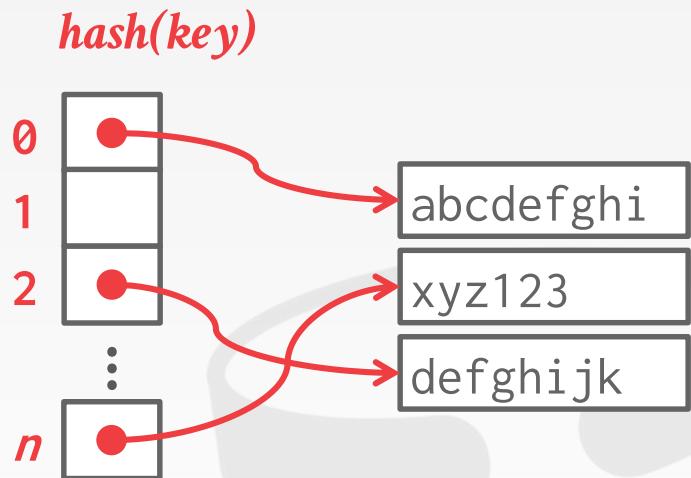
To find an entry, mod the key by the number of elements to find the offset in the array.



STATIC HASH TABLE

Allocate a giant array that has one slot for every element you need to store.

To find an entry, mod the key by the number of elements to find the offset in the array.



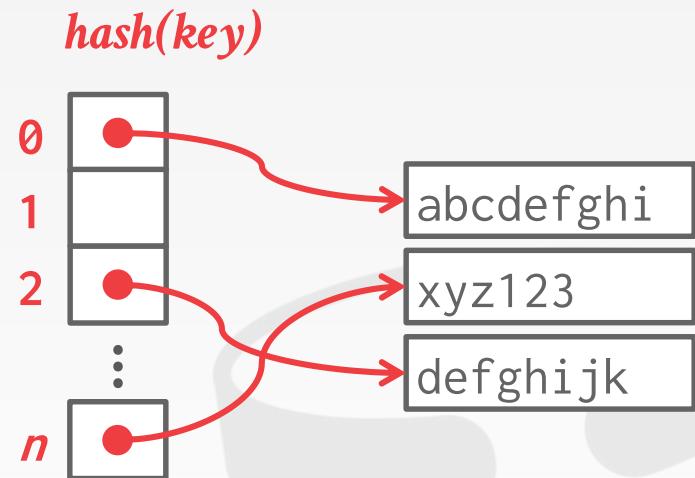
ASSUMPTIONS

You know the number of elements ahead of time.

Each key is unique.

Perfect hash function.

→ If $\text{key}_1 \neq \text{key}_2$, then
 $\text{hash}(\text{key}_1) \neq \text{hash}(\text{key}_2)$



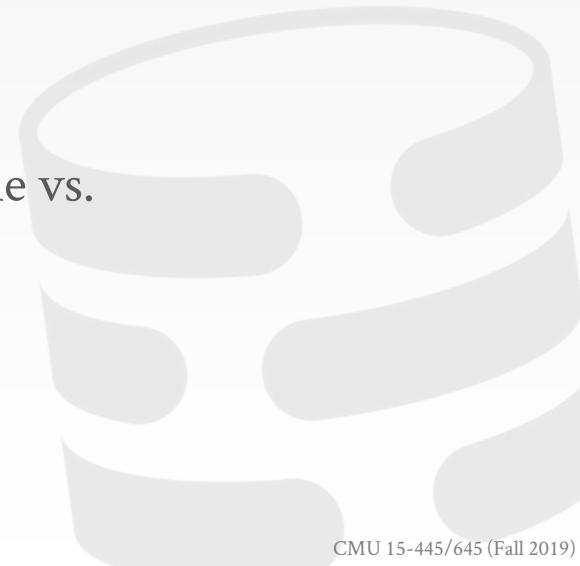
HASH TABLE

Design Decision #1: Hash Function

- How to map a large key space into a smaller domain.
- Trade-off between being fast vs. collision rate.

Design Decision #2: Hashing Scheme

- How to handle key collisions after hashing.
- Trade-off between allocating a large hash table vs. additional instructions to find/insert keys.



TODAY'S AGENDA

Hash Functions

Static Hashing Schemes

Dynamic Hashing Schemes

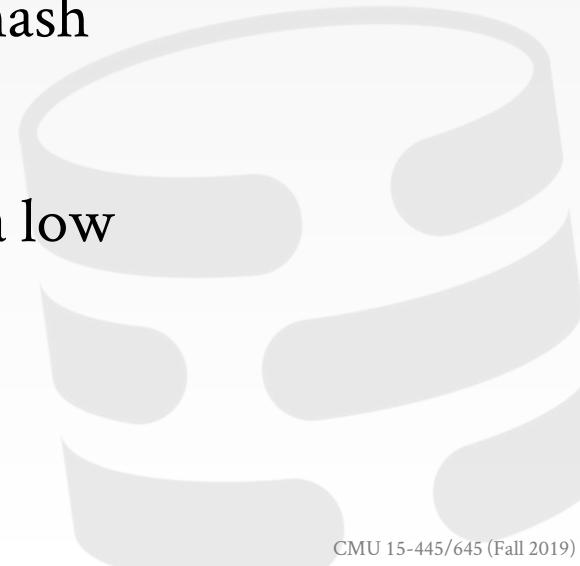


HASH FUNCTIONS

For any input key, return an integer representation of that key.

We do not want to use a cryptographic hash function for DBMS hash tables.

We want something that is fast and has a low collision rate.



HASH FUNCTIONS

CRC-64 (1975)

→ Used in networking for error detection.

MurmurHash (2008)

→ Designed to a fast, general purpose hash function.

Google CityHash (2011)

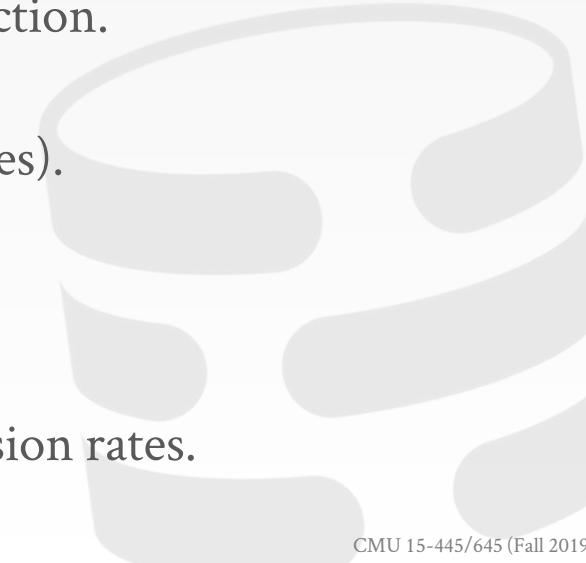
→ Designed to be faster for short keys (<64 bytes).

Facebook XXHash (2012)

→ From the creator of zstd compression.

Google FarmHash (2014)

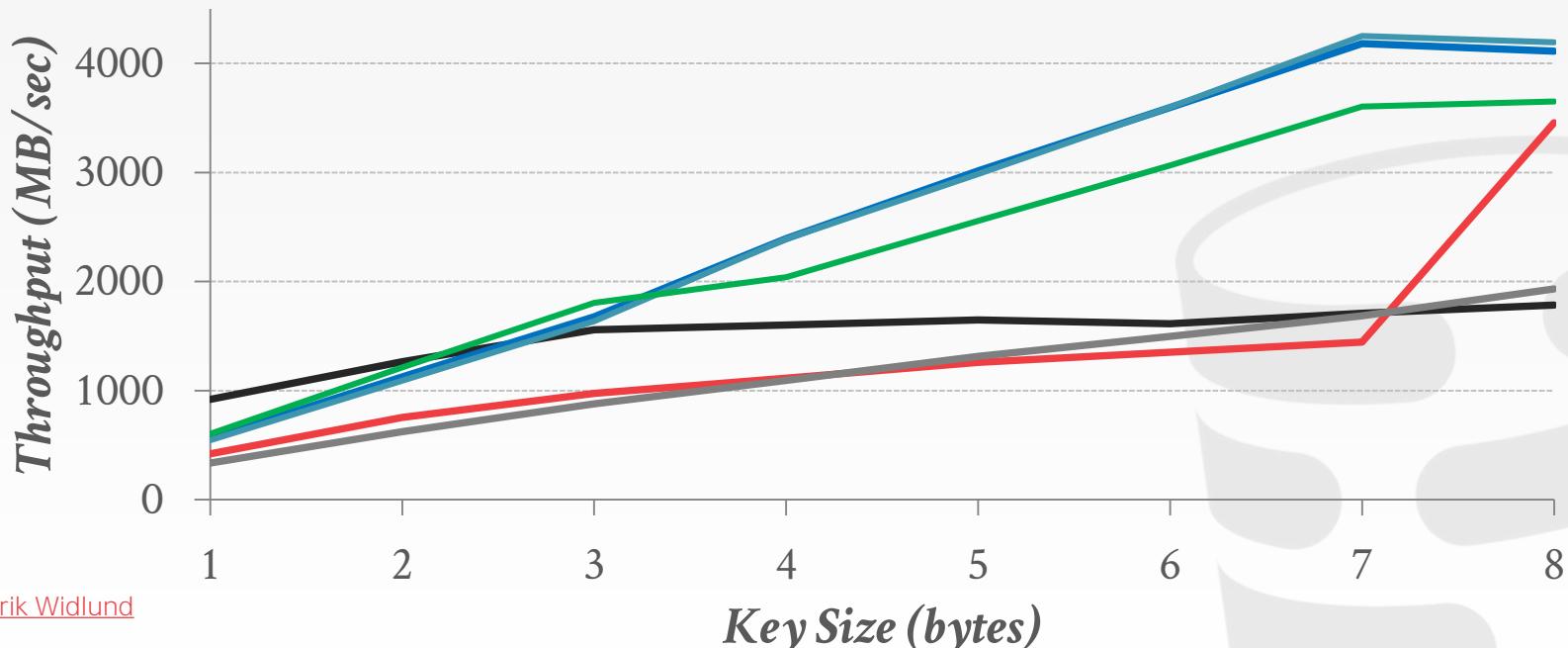
→ Newer version of CityHash with better collision rates.



HASH FUNCTION BENCHMARK

Intel Core i7-8700K @ 3.70GHz

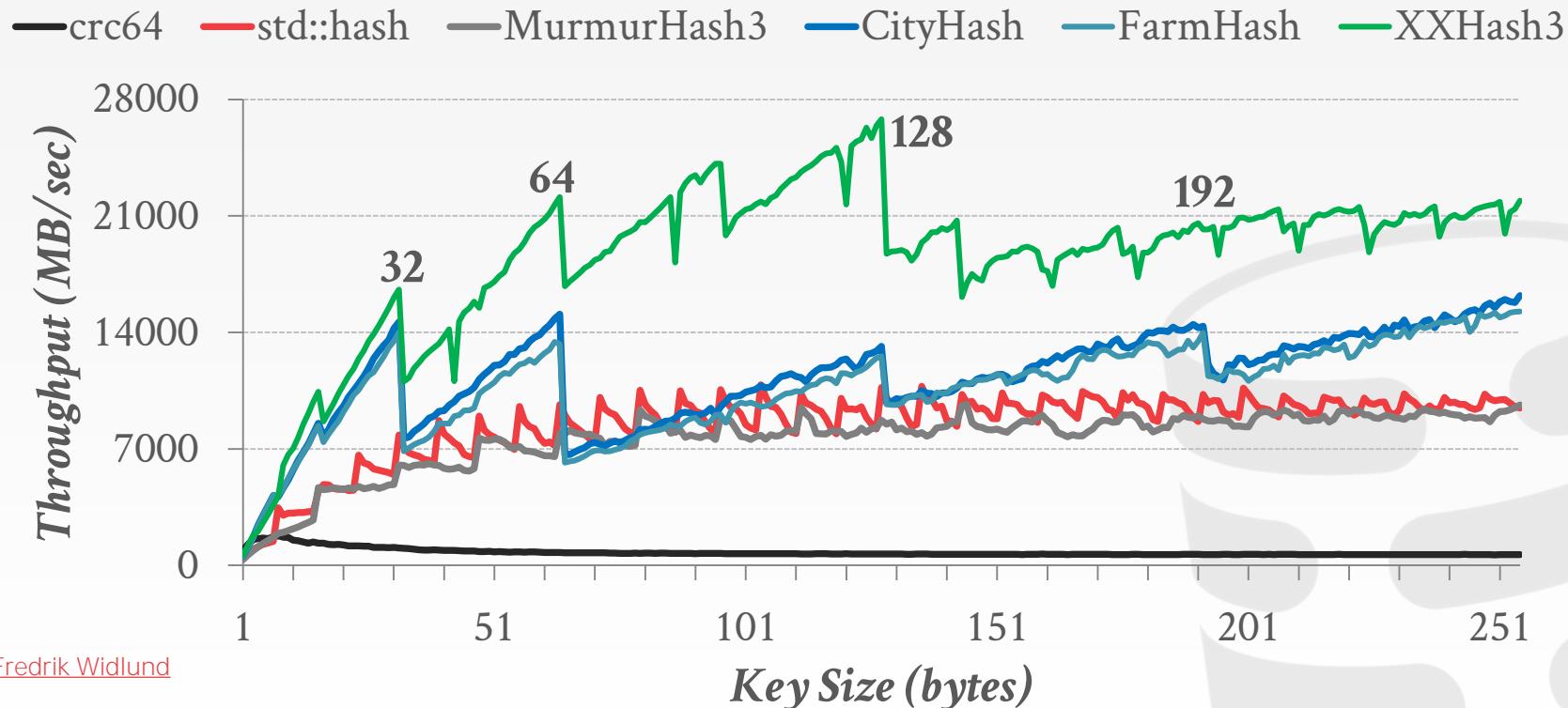
—crc64 —std::hash —MurmurHash3 —CityHash —FarmHash —XXHash3



Source: [Fredrik Widlund](#)

HASH FUNCTION BENCHMARK

Intel Core i7-8700K @ 3.70GHz



STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Robin Hood Hashing

Approach #3: Cuckoo Hashing



LINEAR PROBE HASHING

Single giant table of slots.

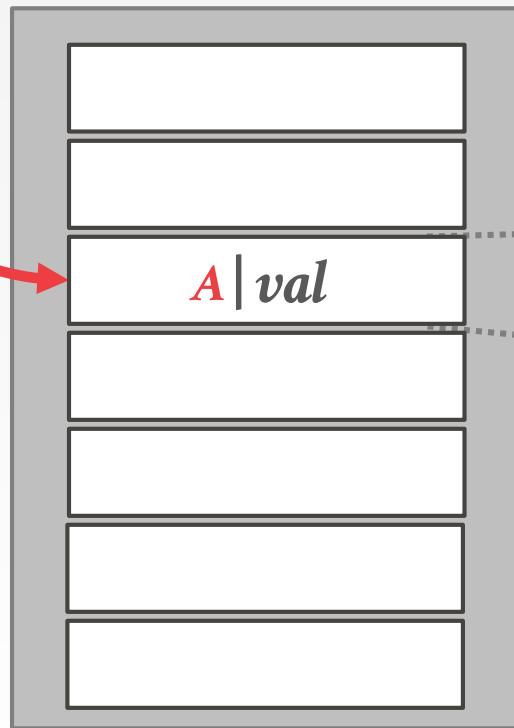
Resolve collisions by linearly searching for the next free slot in the table.

- To determine whether an element is present, hash to a location in the index and scan for it.
- Have to store the key in the index to know when to stop scanning.
- Insertions and deletions are generalizations of lookups.

LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

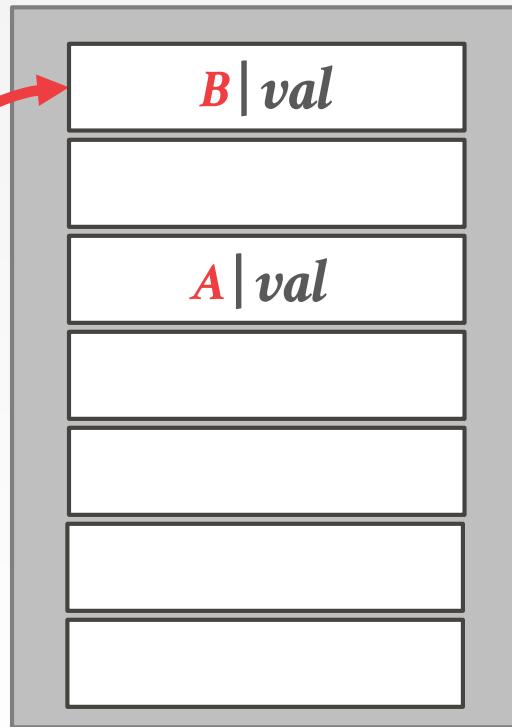


<key> | <value>

LINEAR PROBE HASHING

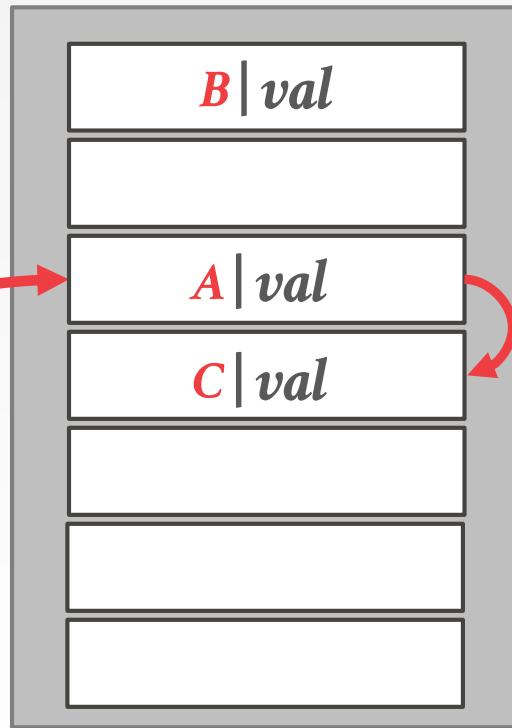
hash(key)

A
B
C
D
E
F



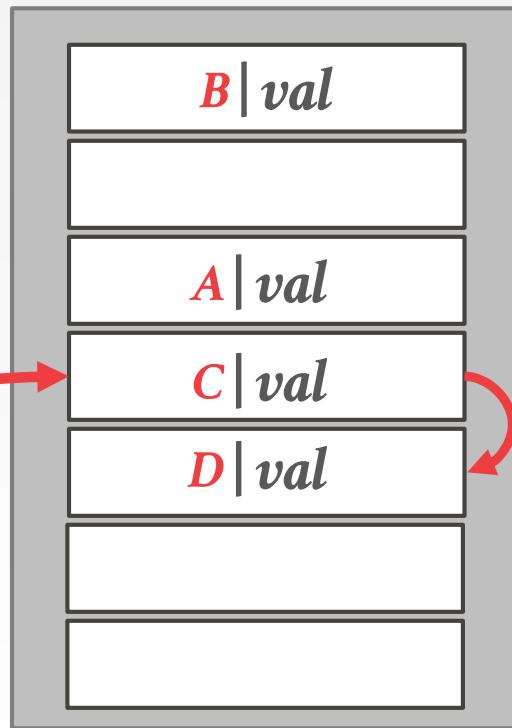
LINEAR PROBE HASHING

hash(key)



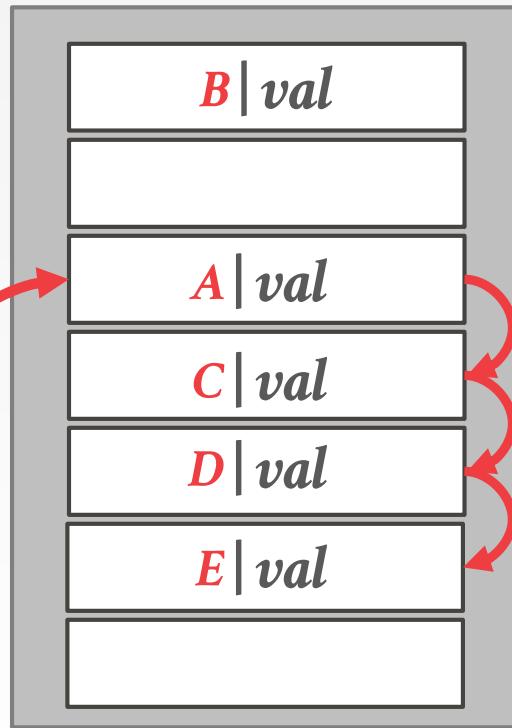
LINEAR PROBE HASHING

hash(key)



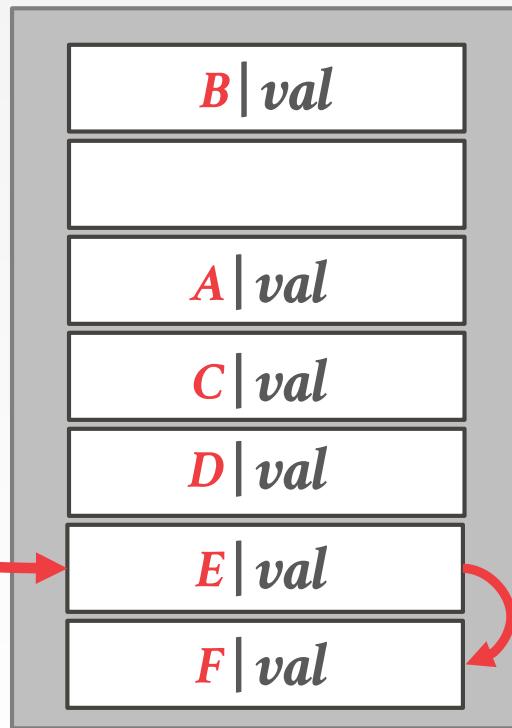
LINEAR PROBE HASHING

hash(key)

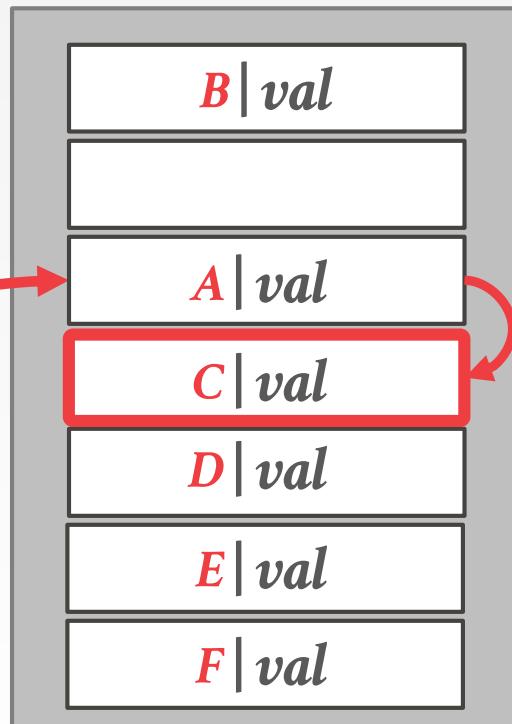
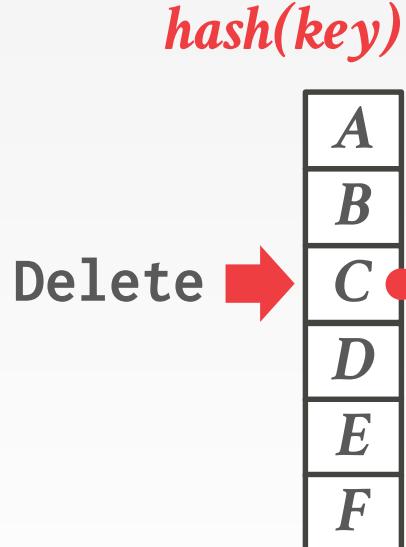


LINEAR PROBE HASHING

hash(key)



LINEAR PROBE HASHING – DELETES

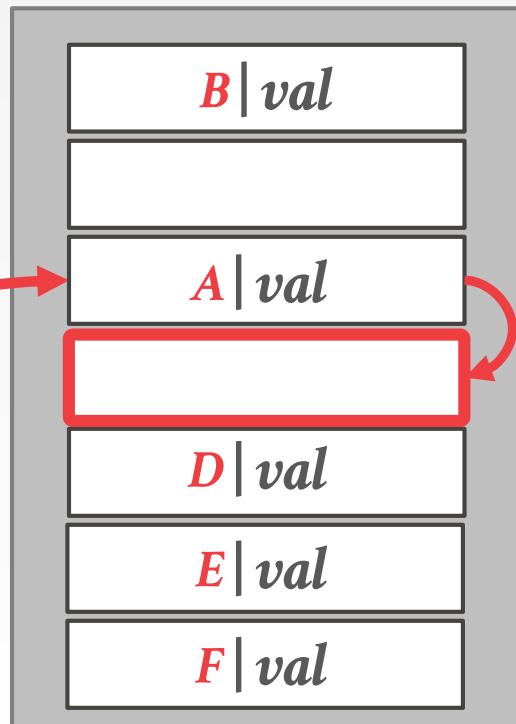


LINEAR PROBE HASHING – DELETES

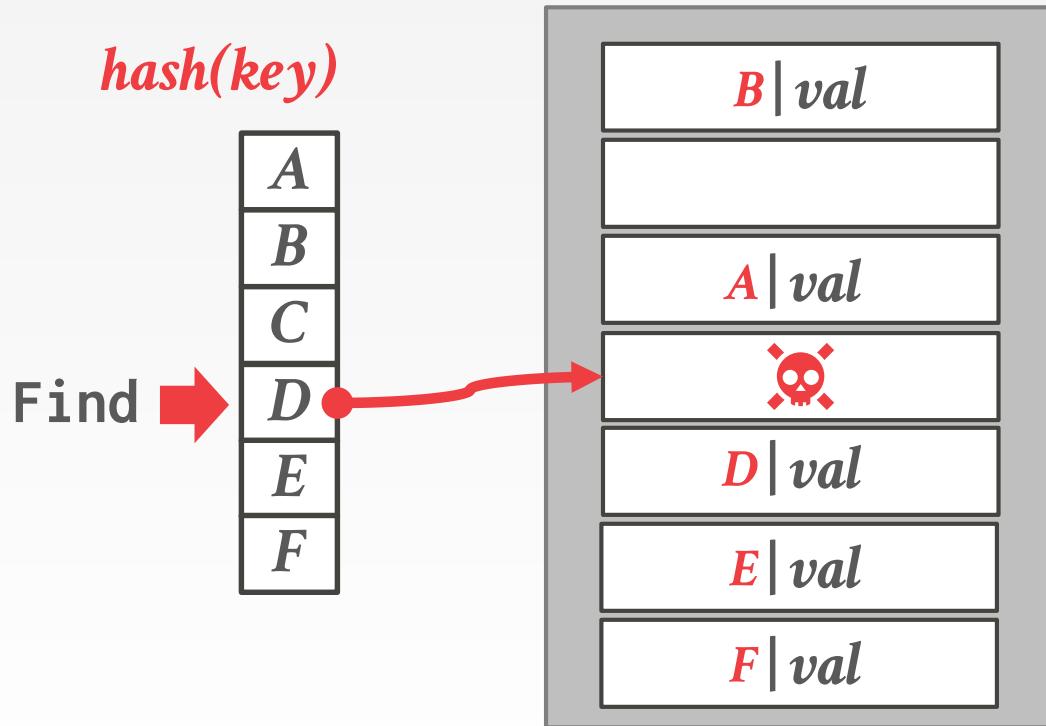
hash(key)

Delete

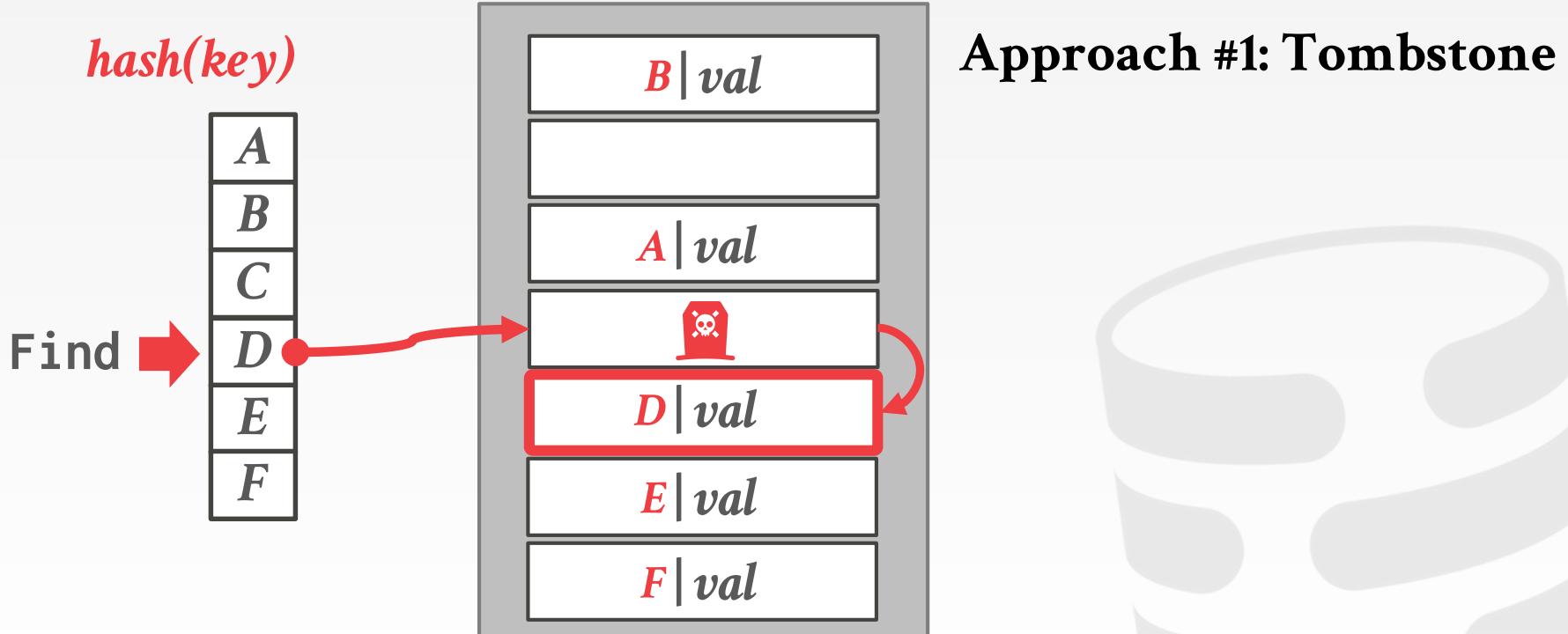
A
B
C
D
E
F



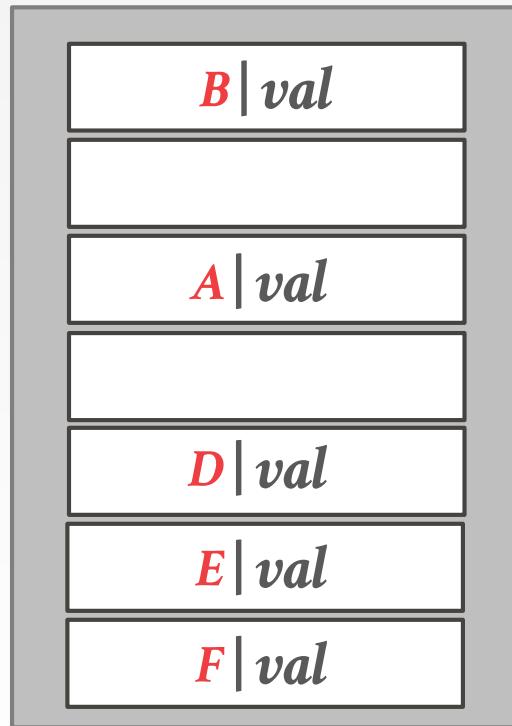
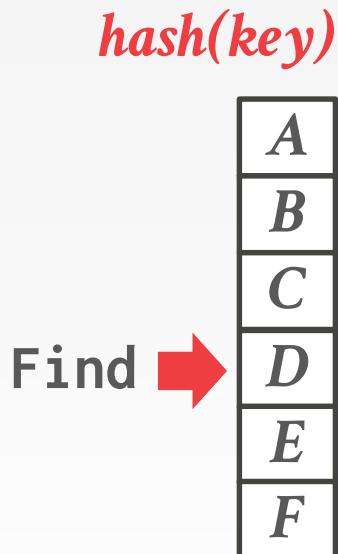
LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES

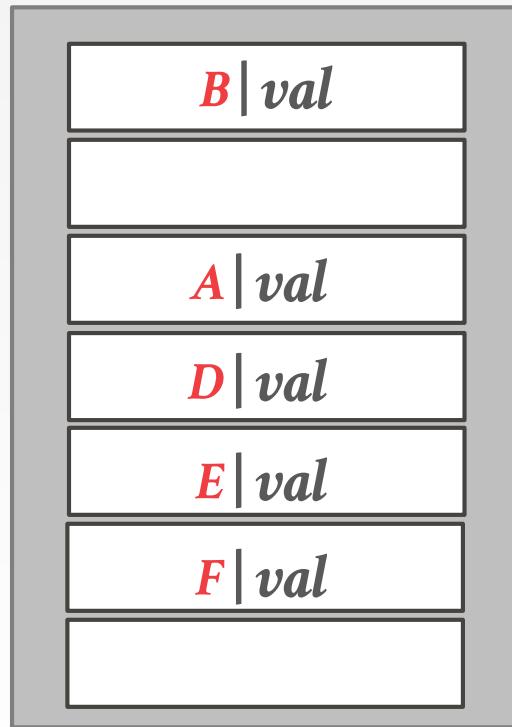
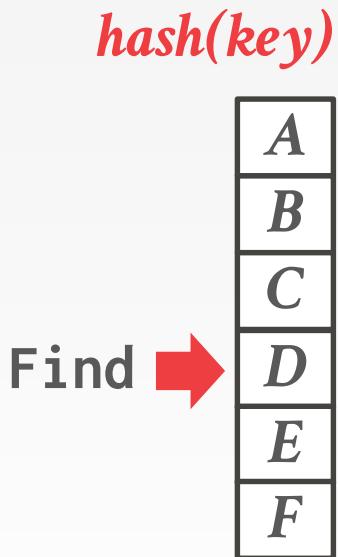


Approach #1: Tombstone

Approach #2: Movement



LINEAR PROBE HASHING – DELETES

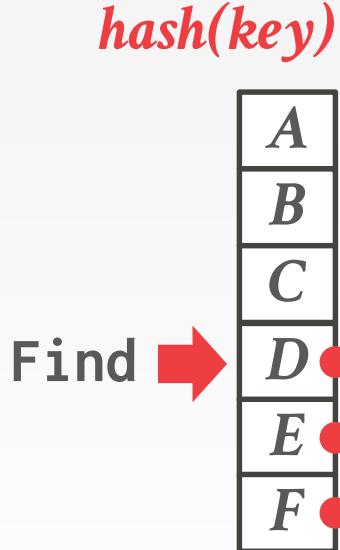


Approach #1: Tombstone

Approach #2: Movement



LINEAR PROBE HASHING – DELETES

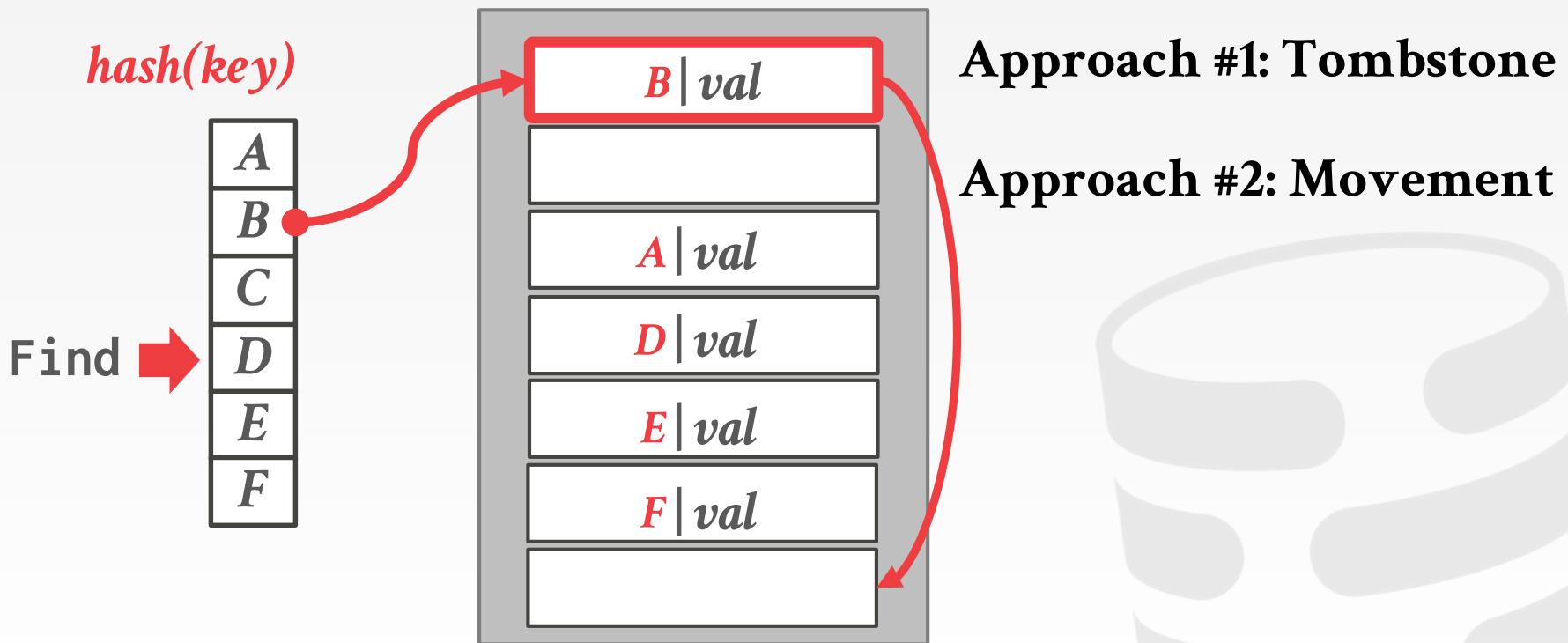


Approach #1: Tombstone

Approach #2: Movement



LINEAR PROBE HASHING – DELETES



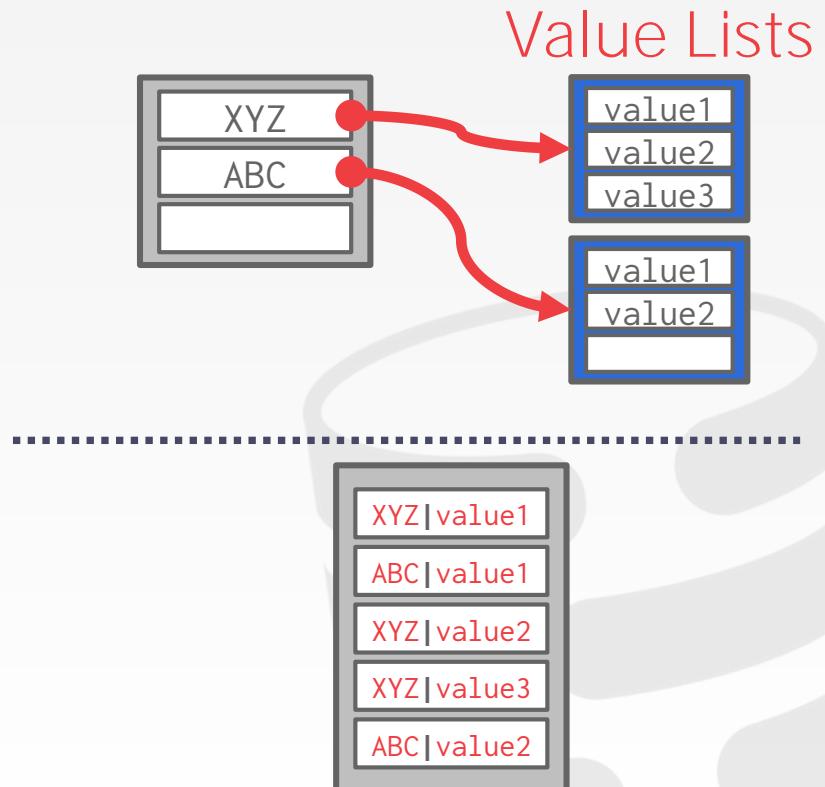
NON-UNIQUE KEYS

Choice #1: Separate Linked List

- Store values in separate storage area for each key.

Choice #2: Redundant Keys

- Store duplicate keys entries together in the hash table.



ROBIN HOOD HASHING

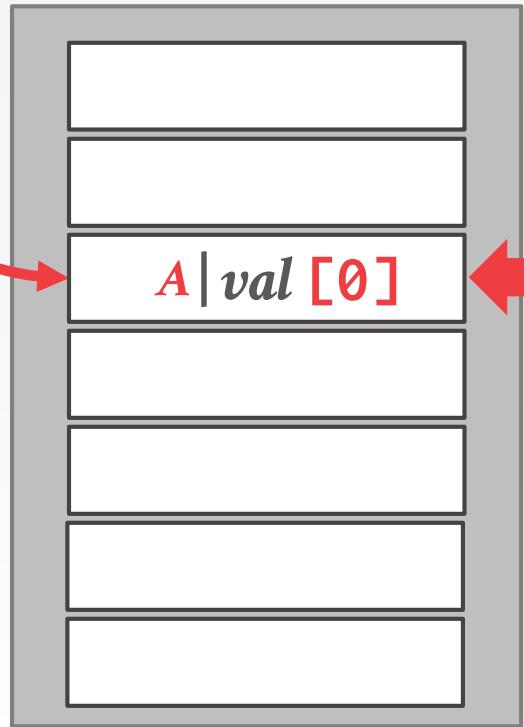
Variant of linear probe hashing that steals slots from "rich" keys and give them to "poor" keys.

- Each key tracks the number of positions they are from where its optimal position in the table.
- On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.

ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

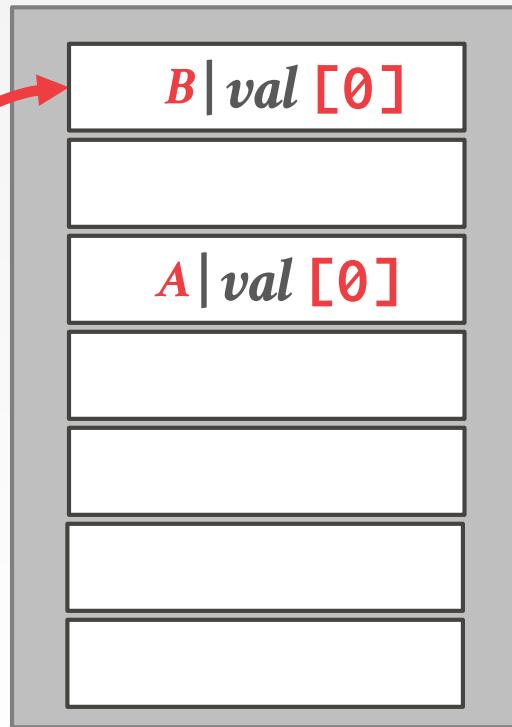


of "Jumps" From First Position

ROBIN HOOD HASHING

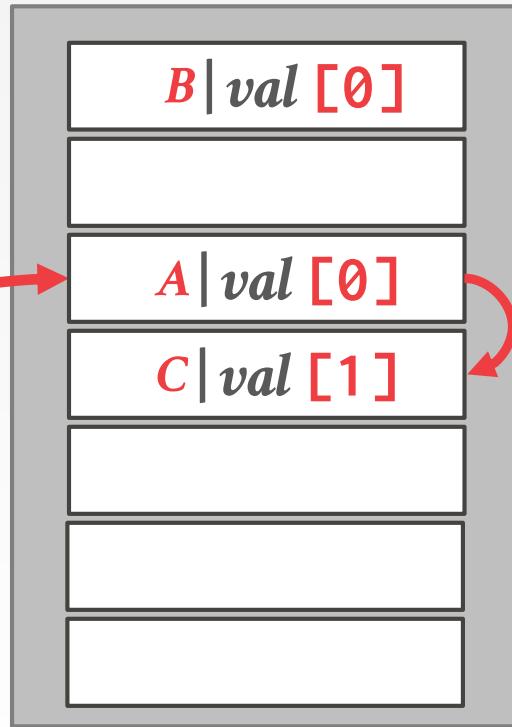
hash(key)

A
B
C
D
E
F



ROBIN HOOD HASHING

hash(key)

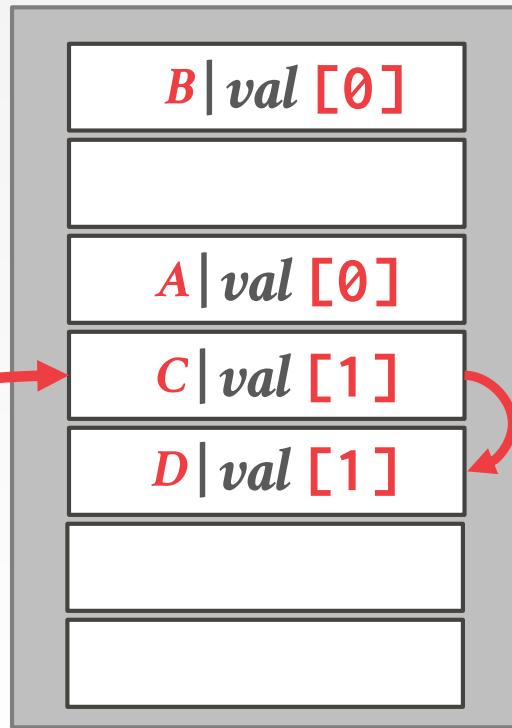


$A[0] == C[0]$



ROBIN HOOD HASHING

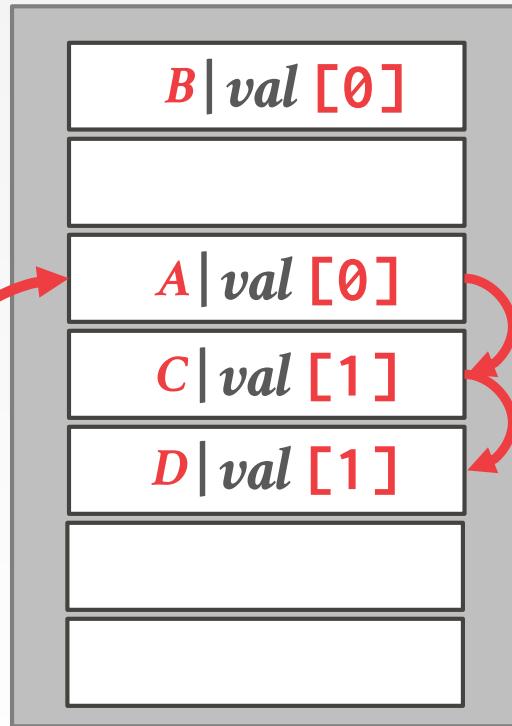
hash(key)



C[1]>D[0]

ROBIN HOOD HASHING

hash(key)



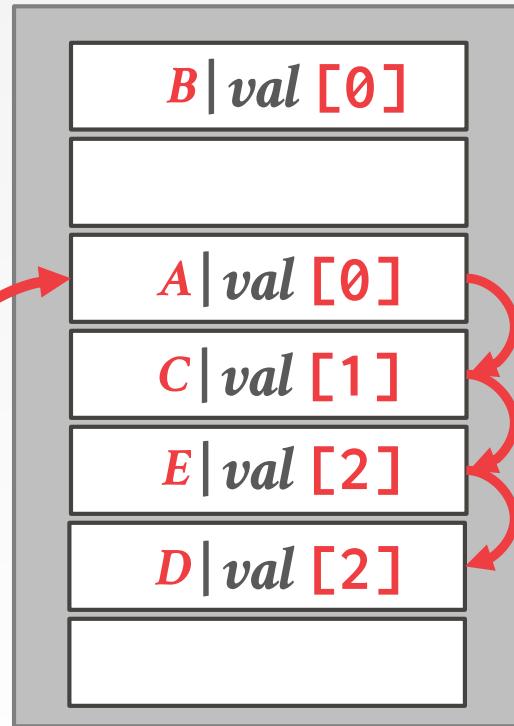
$$A[0] == E[0]$$

$$C[1] == E[1]$$

$$D[1] < E[2]$$

ROBIN HOOD HASHING

hash(key)



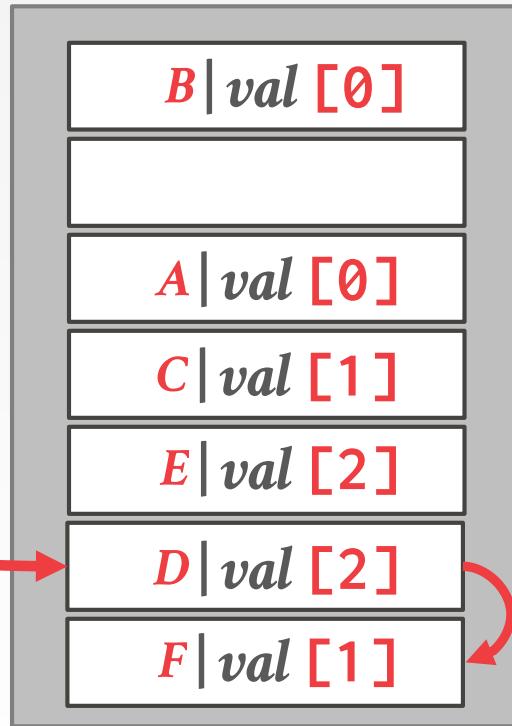
$A[0] == E[0]$

$C[1] == E[1]$

$D[1] < E[2]$

ROBIN HOOD HASHING

hash(key)



$D[2] > F[0]$

CUCKOO HASHING

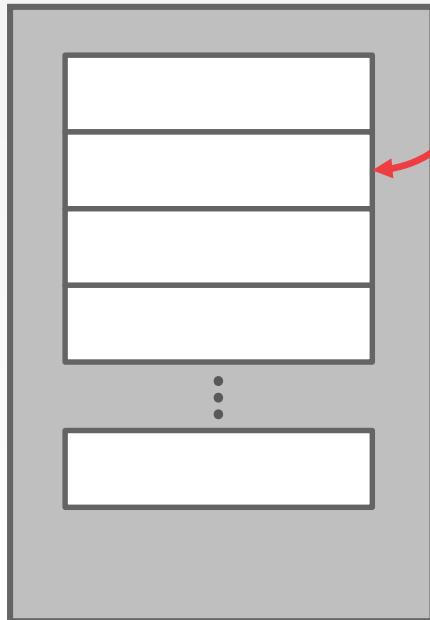
Use multiple hash tables with different hash function seeds.

- On insert, check every table and pick anyone that has a free slot.
- If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

Look-ups and deletions are always **O(1)** because only one location per hash table is checked.

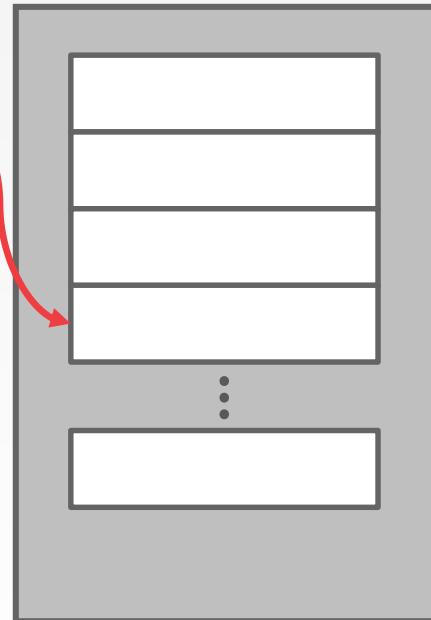
CUCKOO HASHING

Hash Table #1



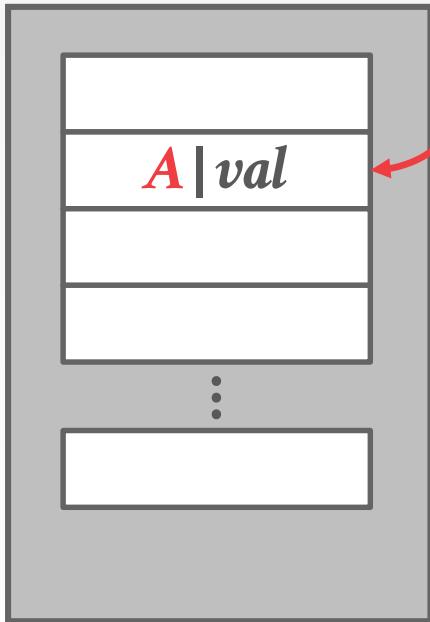
Insert A
 $hash_1(A)$ $hash_2(A)$

Hash Table #2



CUCKOO HASHING

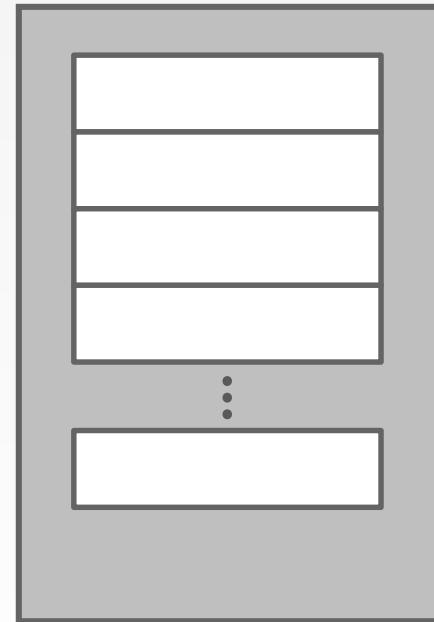
Hash Table #1



Insert A

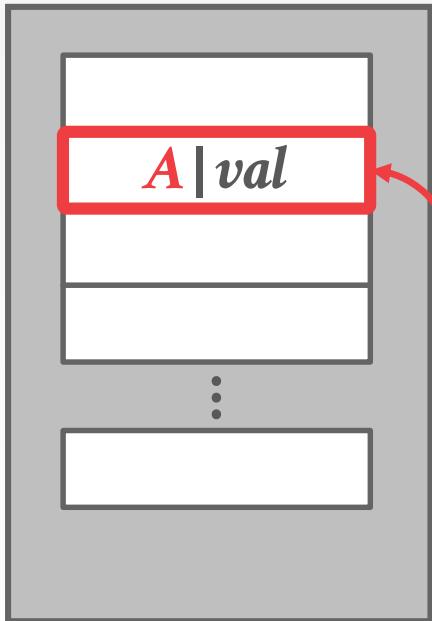
$hash_1(A)$ $hash_2(A)$

Hash Table #2



CUCKOO HASHING

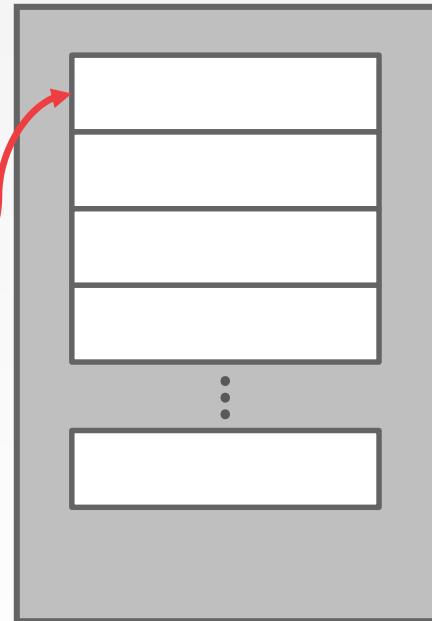
Hash Table #1



Insert A
 $hash_1(A)$ $hash_2(A)$

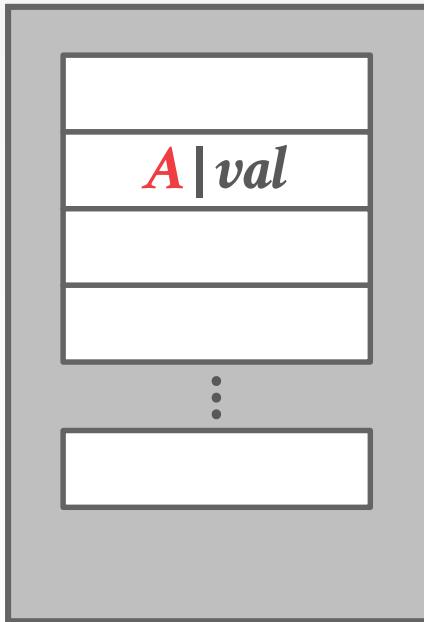
Insert B
 $hash_1(B)$ $hash_2(B)$

Hash Table #2



CUCKOO HASHING

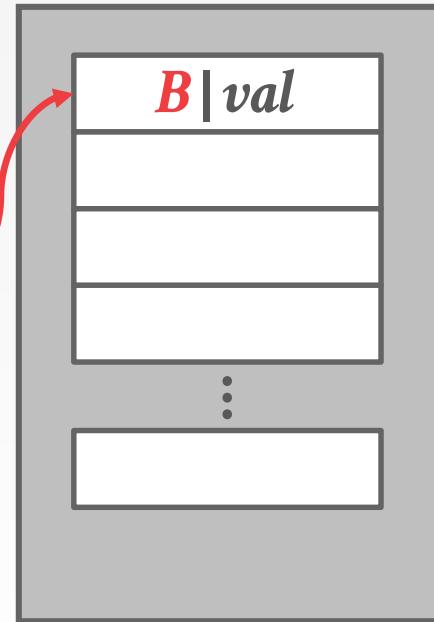
Hash Table #1



Hash Table #2

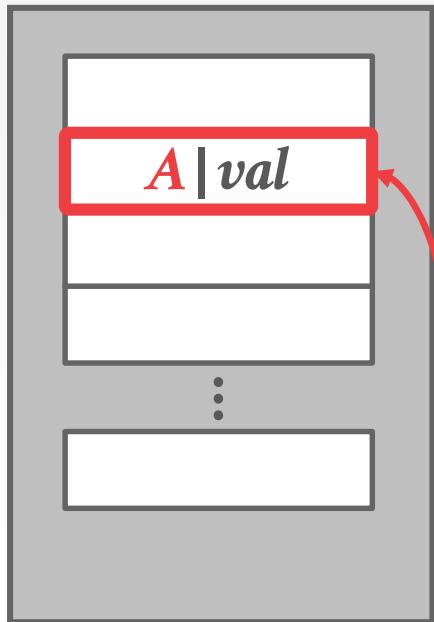
Insert A
 $hash_1(A)$ $hash_2(A)$

Insert B
 $hash_1(B)$ $hash_2(B)$



CUCKOO HASHING

Hash Table #1

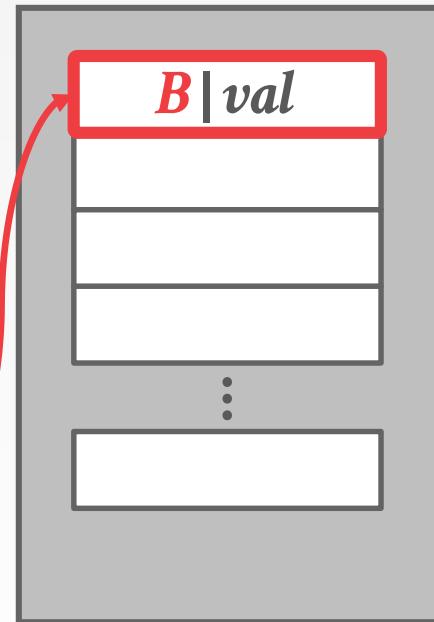


Insert A
 $hash_1(A)$ $hash_2(A)$

Insert B
 $hash_1(B)$ $hash_2(B)$

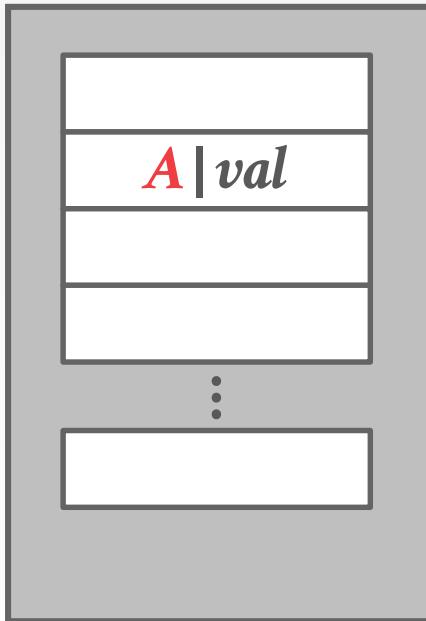
Insert C
 $hash_1(C)$ $hash_2(C)$

Hash Table #2



CUCKOO HASHING

Hash Table #1

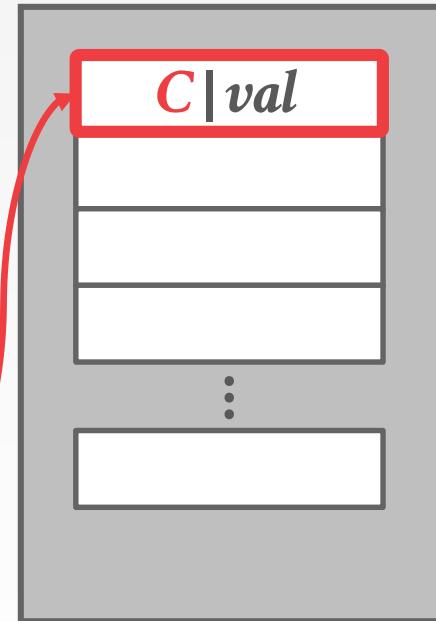


Insert A
 $hash_1(A)$ $hash_2(A)$

Insert B
 $hash_1(B)$ $hash_2(B)$

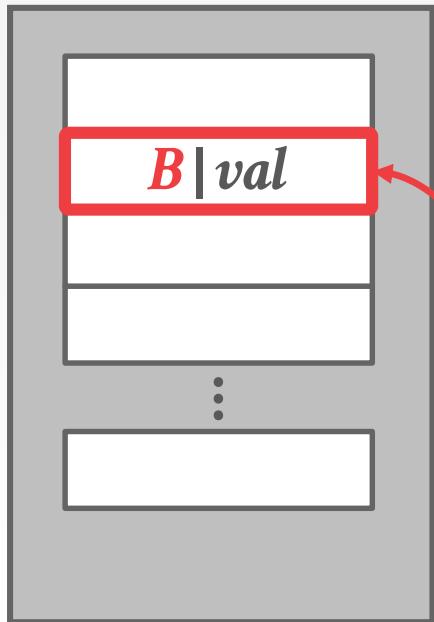
Insert C
 $hash_1(C)$ $hash_2(C)$

Hash Table #2



CUCKOO HASHING

Hash Table #1



Insert A

$hash_1(A)$ $hash_2(A)$

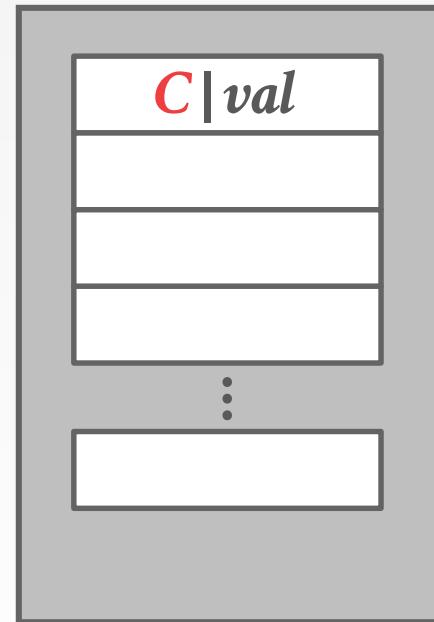
Insert B

$hash_1(B)$ $hash_2(B)$

Insert C

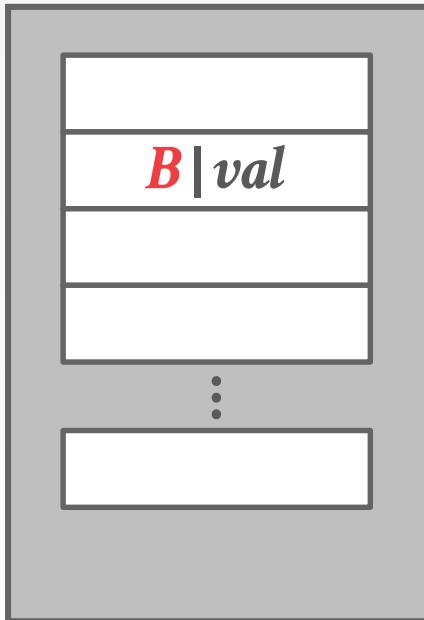
$hash_1(C)$ $hash_2(C)$
 $hash_1(B)$

Hash Table #2



CUCKOO HASHING

Hash Table #1

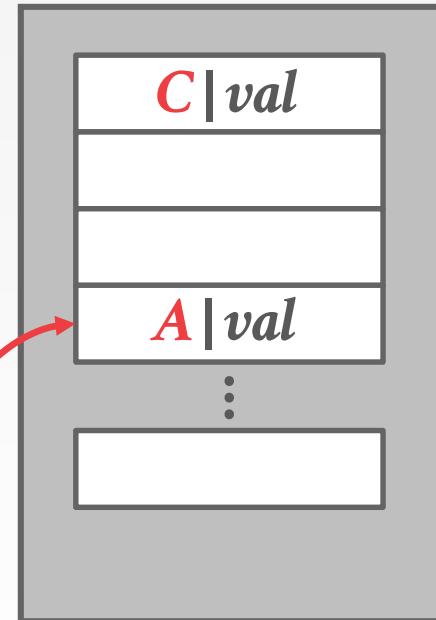


Insert A
 $hash_1(A)$ $hash_2(A)$

Insert B
 $hash_1(B)$ $hash_2(B)$

Insert C
 $hash_1(C)$ $hash_2(C)$
 $hash_1(B)$
 $hash_2(A)$

Hash Table #2



OBSERVATION

The previous hash tables require the DBMS to know the number of elements it wants to store.

- Otherwise it has rebuild the table if it needs to grow/shrink in size.

Dynamic hash tables resize themselves on demand.

- Chained Hashing
- Extendible Hashing
- Linear Hashing

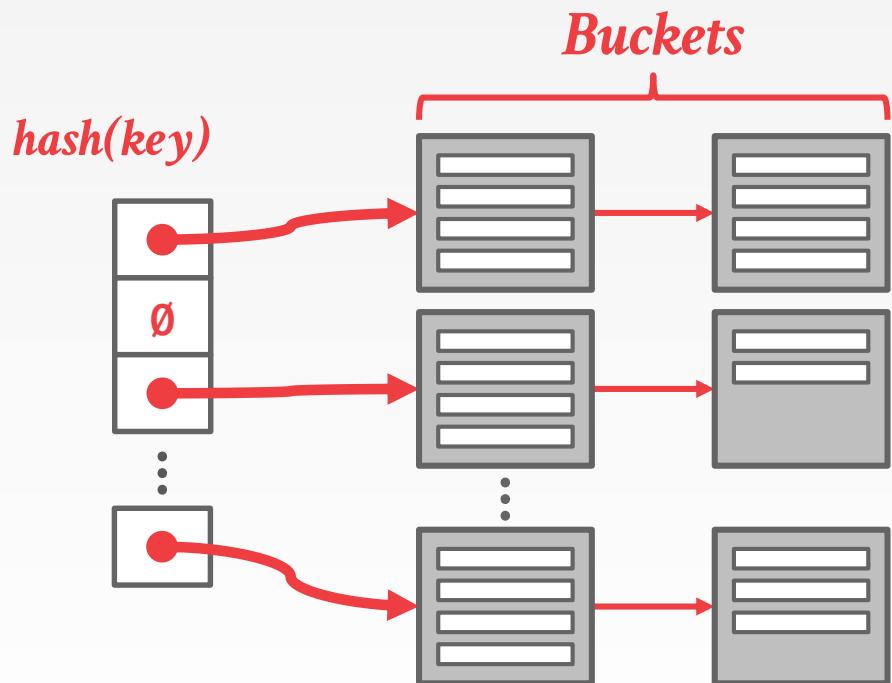
CHAINED HASHING

Maintain a linked list of **buckets** for each slot in the hash table.

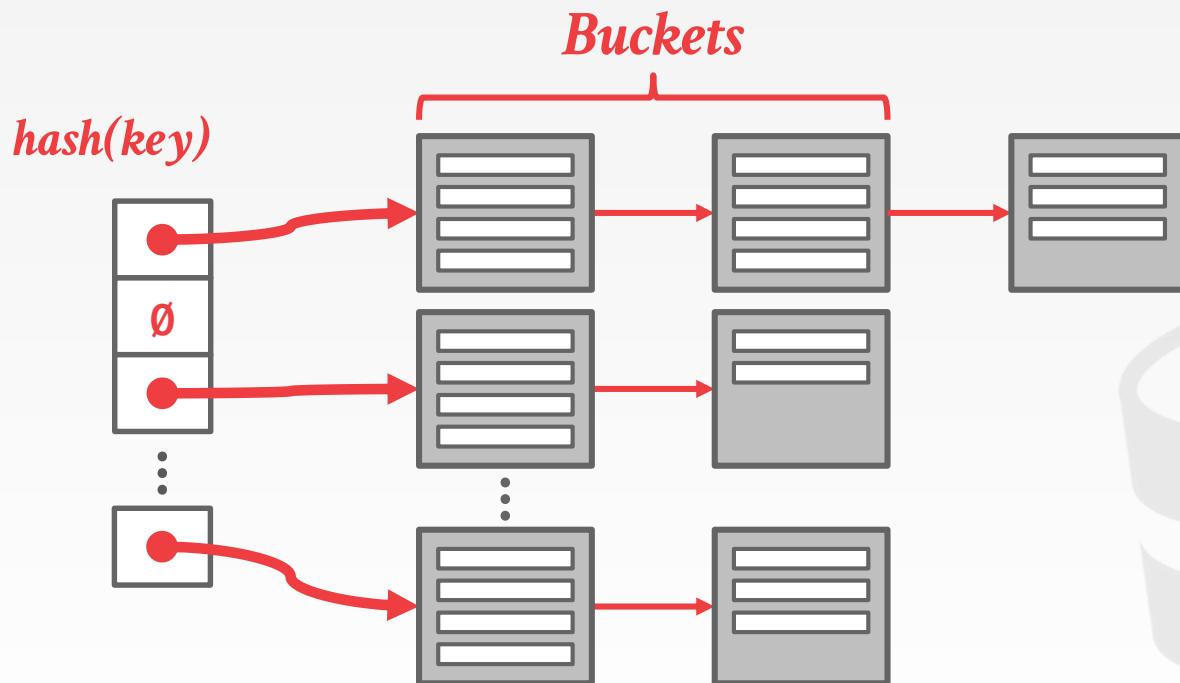
Resolve collisions by placing all elements with the same hash key into the same bucket.

- To determine whether an element is present, hash to its bucket and scan for it.
- Insertions and deletions are generalizations of lookups.

CHAINED HASHING



CHAINED HASHING



EXTENDIBLE HASHING

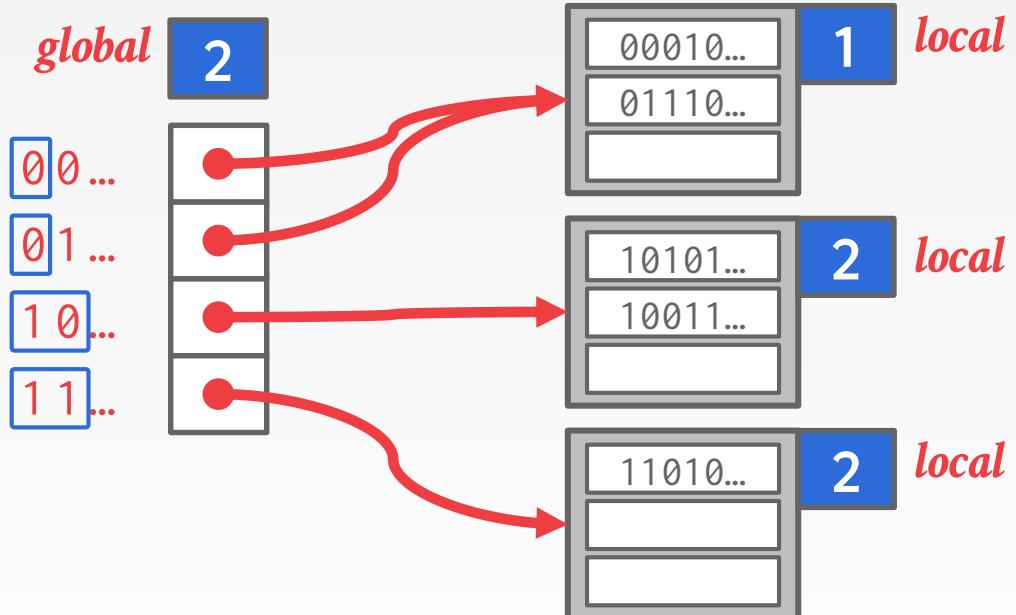
Chained-hashing approach where we split buckets instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

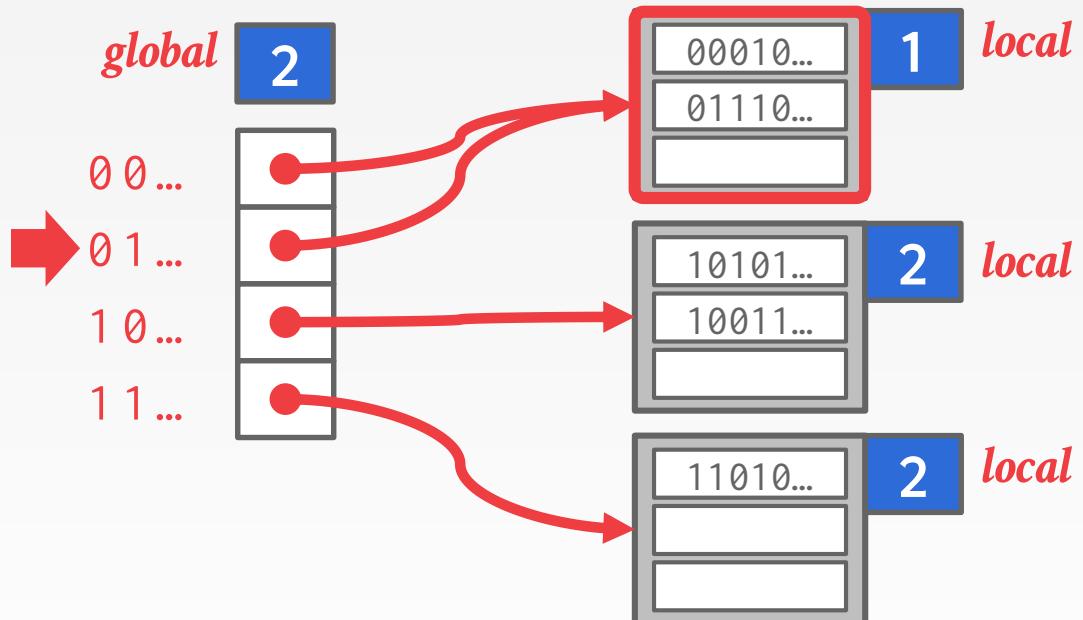
Reshuffling bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.

EXTENDIBLE HASHING



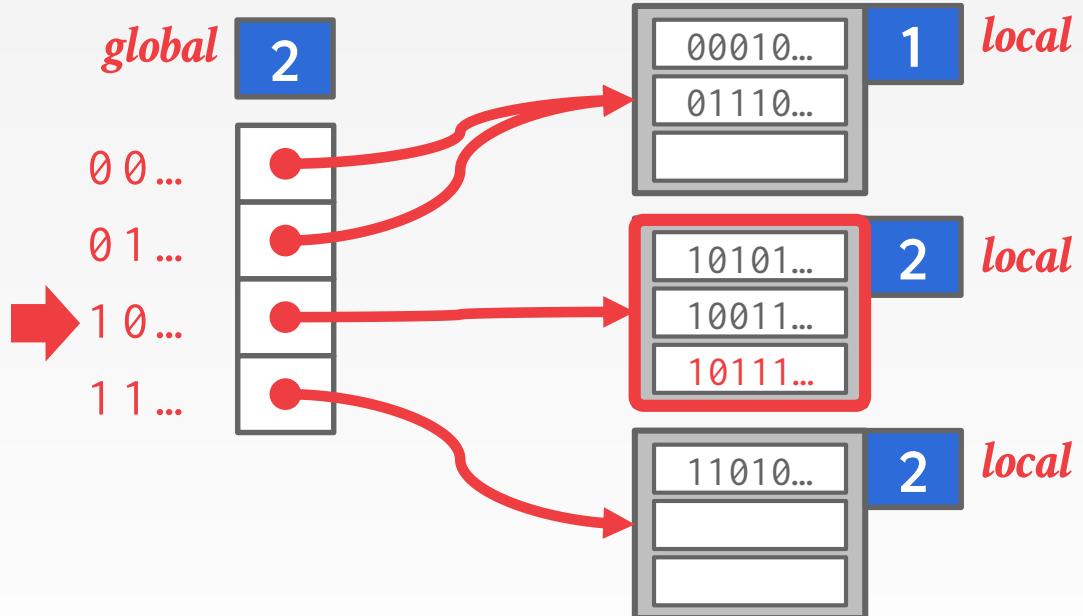
EXTENDIBLE HASHING



Find A
 $hash(A) = 01110...$



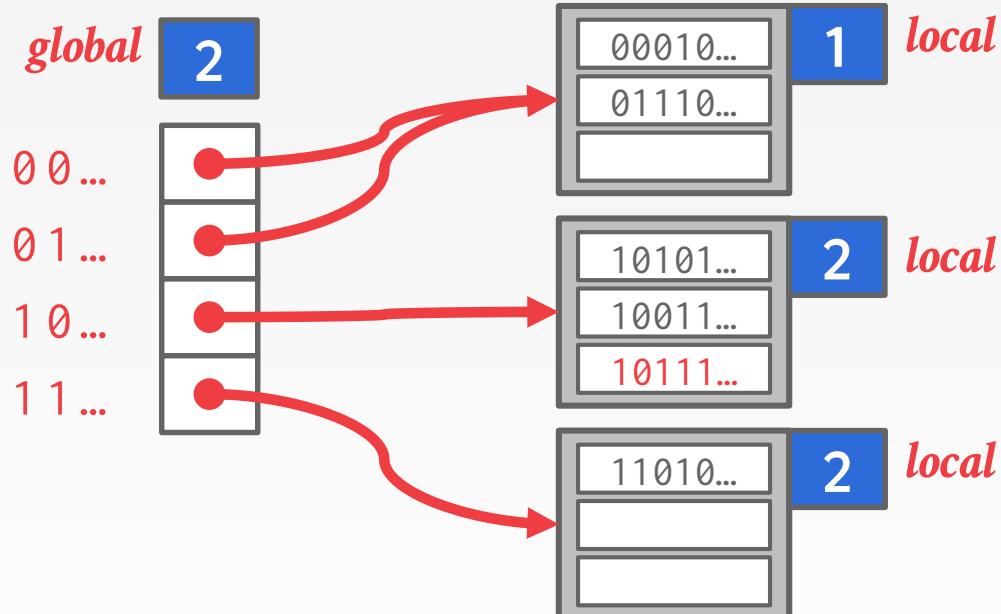
EXTENDIBLE HASHING



Find A
 $\text{hash}(A) = 01110...$

Insert B
 $\text{hash}(B) = \boxed{10}111...$

EXTENDIBLE HASHING

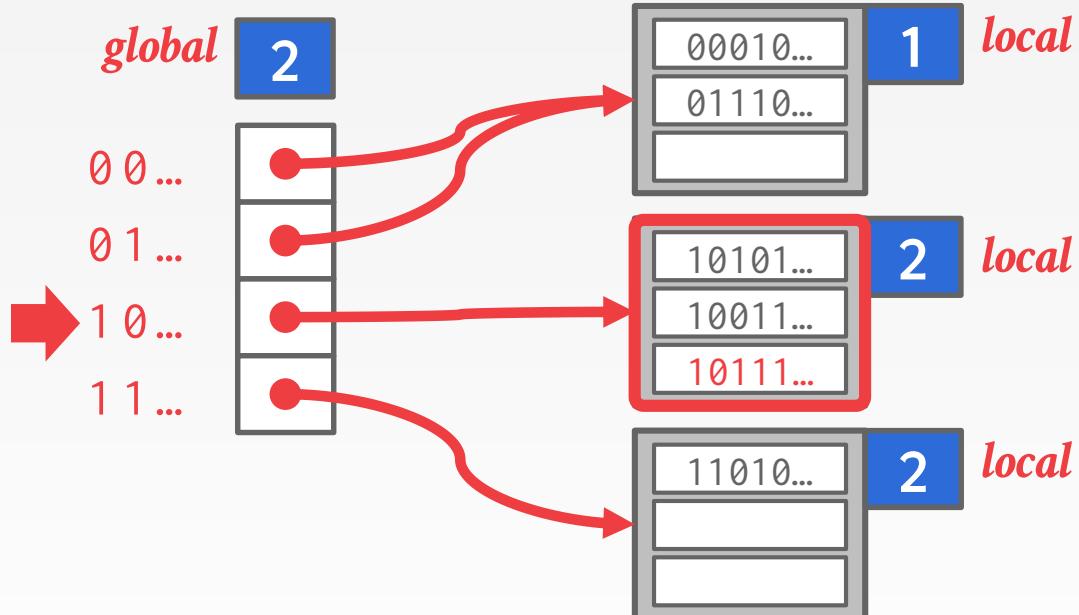


Find A
 $\text{hash}(A) = 01110...$

Insert B
 $\text{hash}(B) = 10111...$

Insert C
 $\text{hash}(C) = \boxed{10100...}$

EXTENDIBLE HASHING

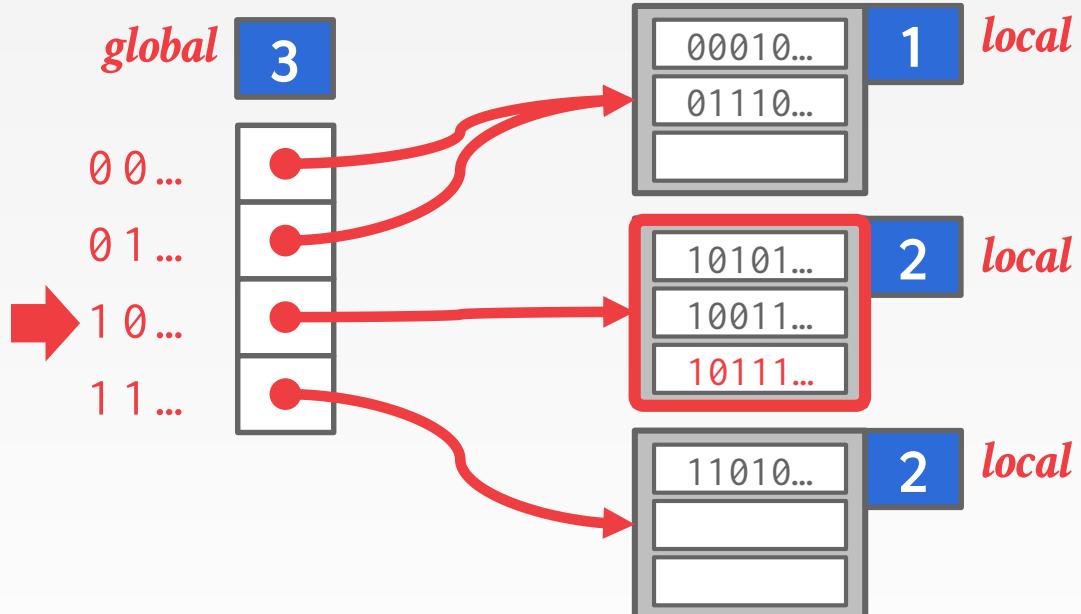


Find A
 $\text{hash}(A) = 01110...$

Insert B
 $\text{hash}(B) = 10111...$

Insert C
 $\text{hash}(C) = \boxed{10100...}$

EXTENDIBLE HASHING

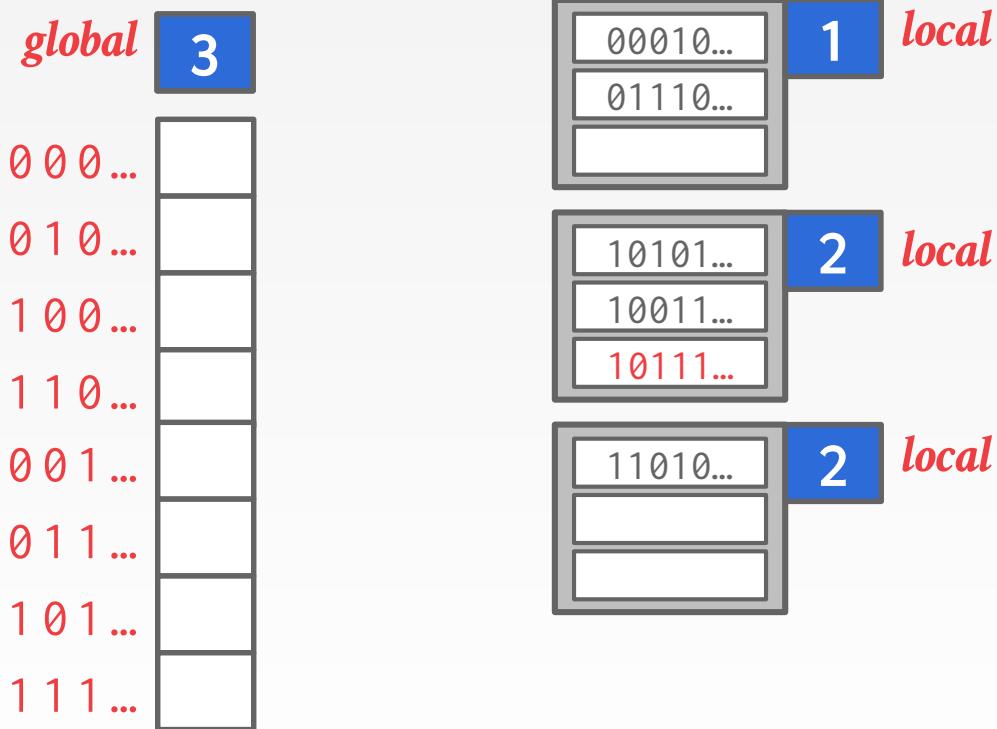


Find A
 $\text{hash}(A) = 01110\dots$

Insert B
 $\text{hash}(B) = 10111\dots$

Insert C
 $\text{hash}(C) = \boxed{10100\dots}$

EXTENDIBLE HASHING

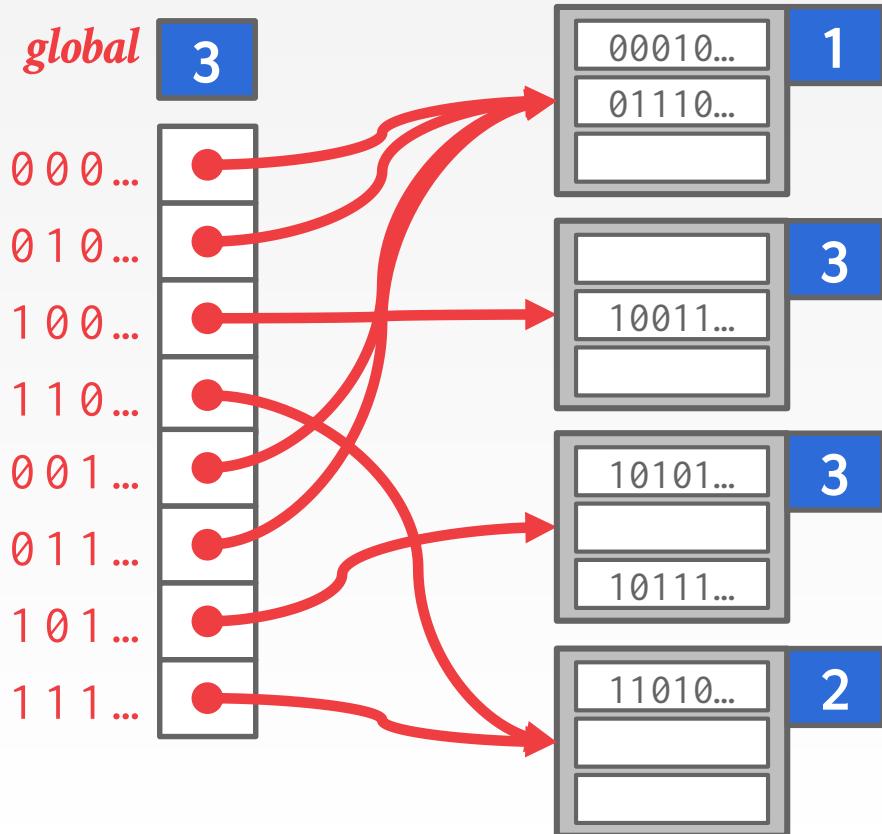


Find A
 $\text{hash}(A) = 01110...$

Insert B
 $\text{hash}(B) = 10111...$

Insert C
 $\text{hash}(C) = \boxed{10100...}$

EXTENDIBLE HASHING



Find A

$\text{hash}(A) = 01110...$

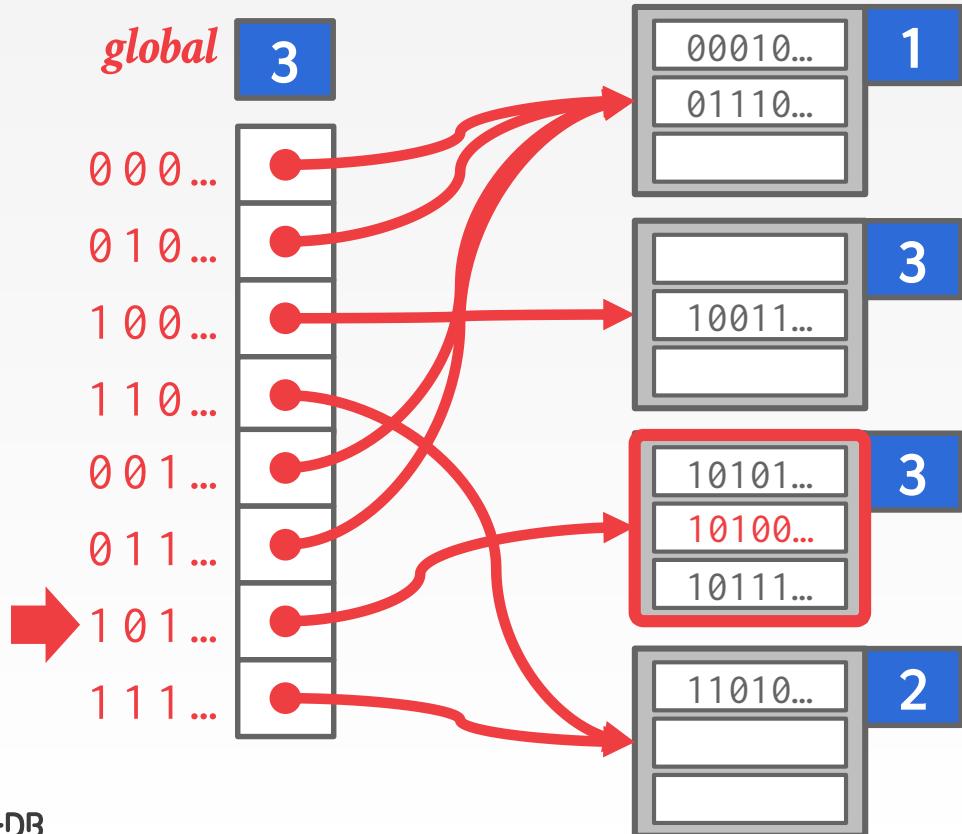
Insert B

$\text{hash}(B) = 10111...$

Insert C

$\text{hash}(C) = 10100...$

EXTENDIBLE HASHING



Find A

$\text{hash}(A) = 01110...$

Insert B

$\text{hash}(B) = 10111...$

Insert C

$\text{hash}(C) = \boxed{10100...}$

LINEAR HASHING

The hash table maintains a pointer that tracks the next bucket to split.

→ When any bucket overflows, split the bucket at the pointer location.

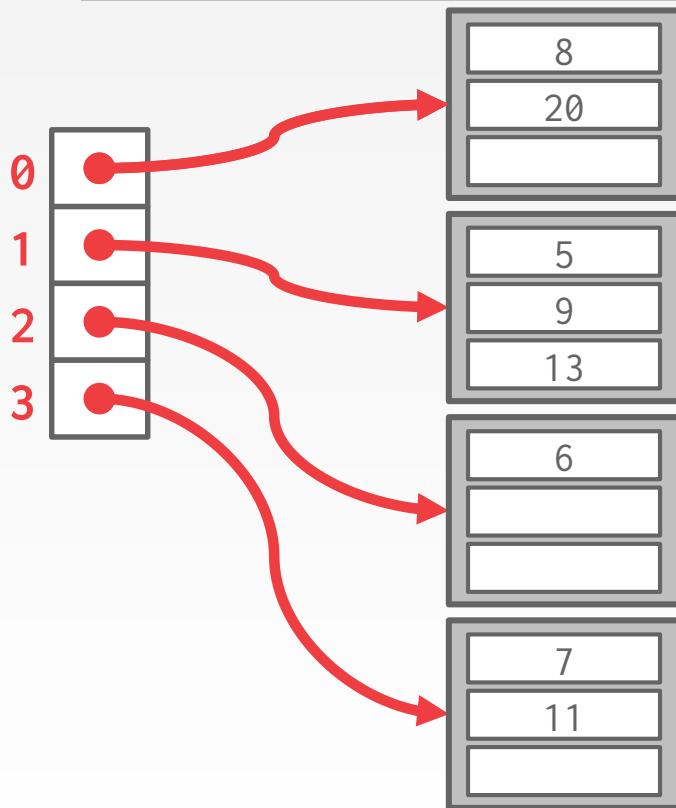
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

→ Space Utilization
→ Average Length of Overflow Chains

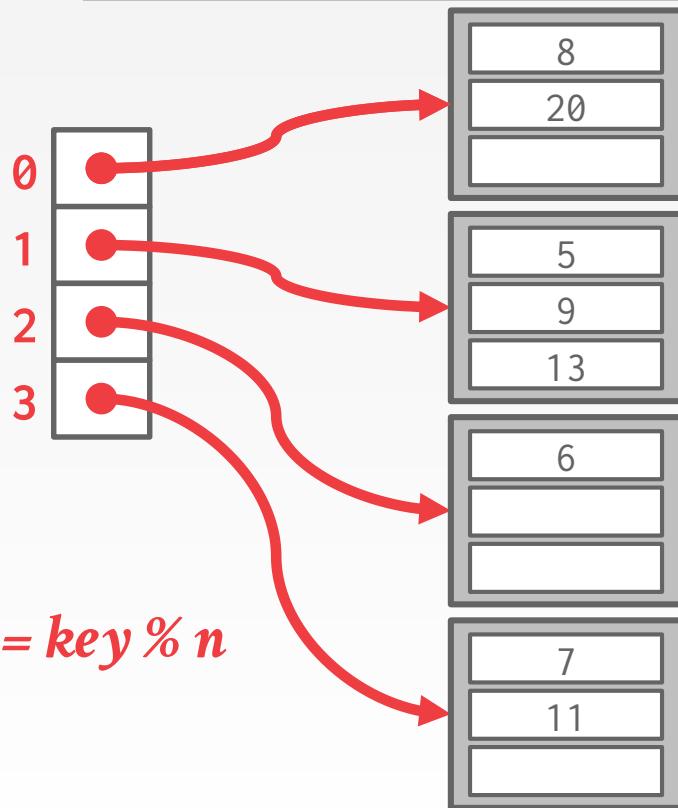


LINEAR HASHING



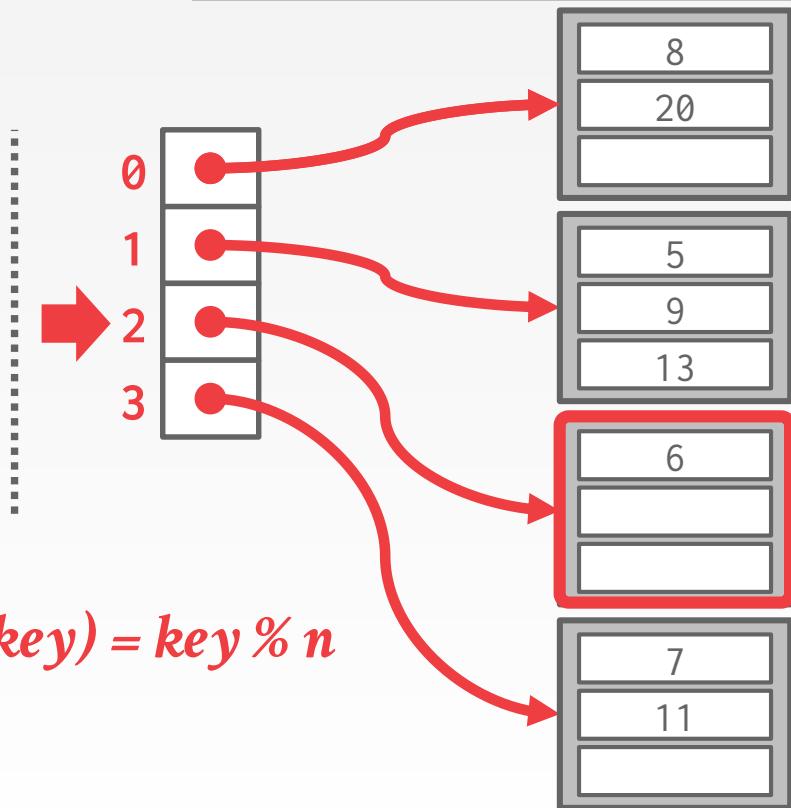
LINEAR HASHING

*Split
Pointer*



LINEAR HASHING

*Split
Pointer*



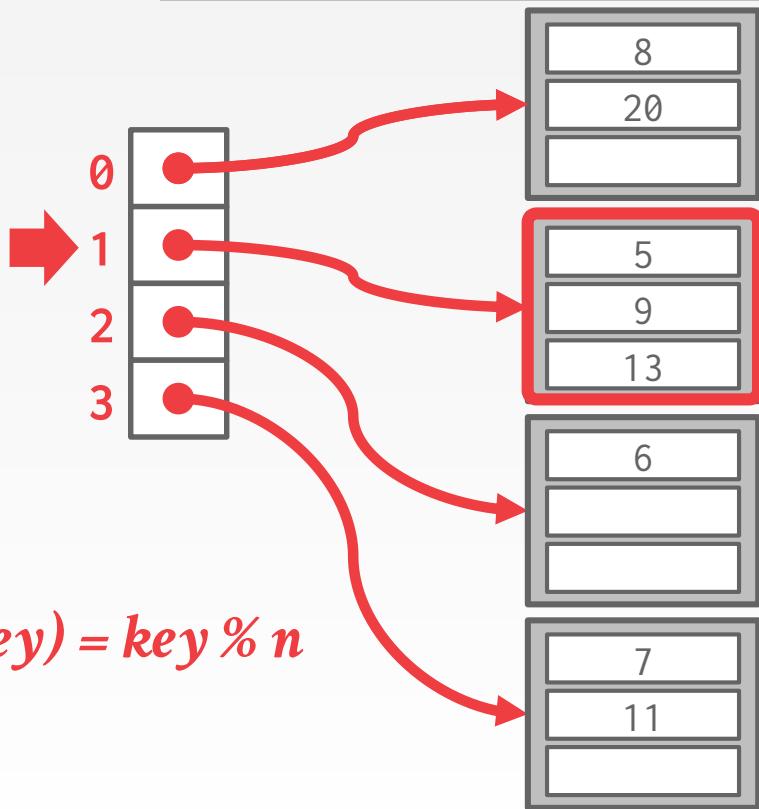
Find 6

$$hash_1(6) = 6 \% 4 = 2$$



LINEAR HASHING

*Split
Pointer*

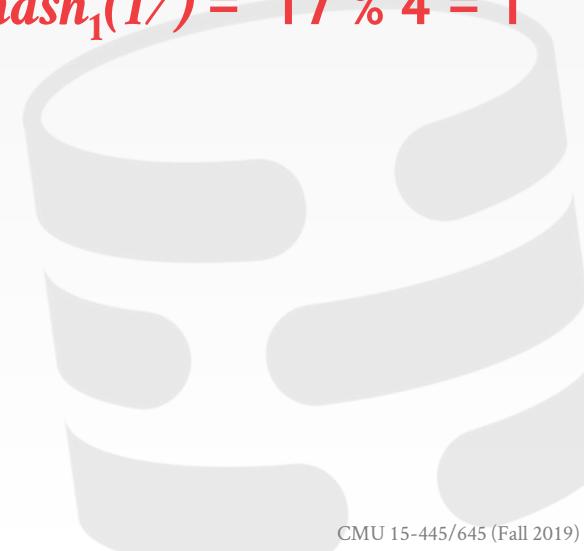


Find 6

$$hash_1(6) = 6 \% 4 = 2$$

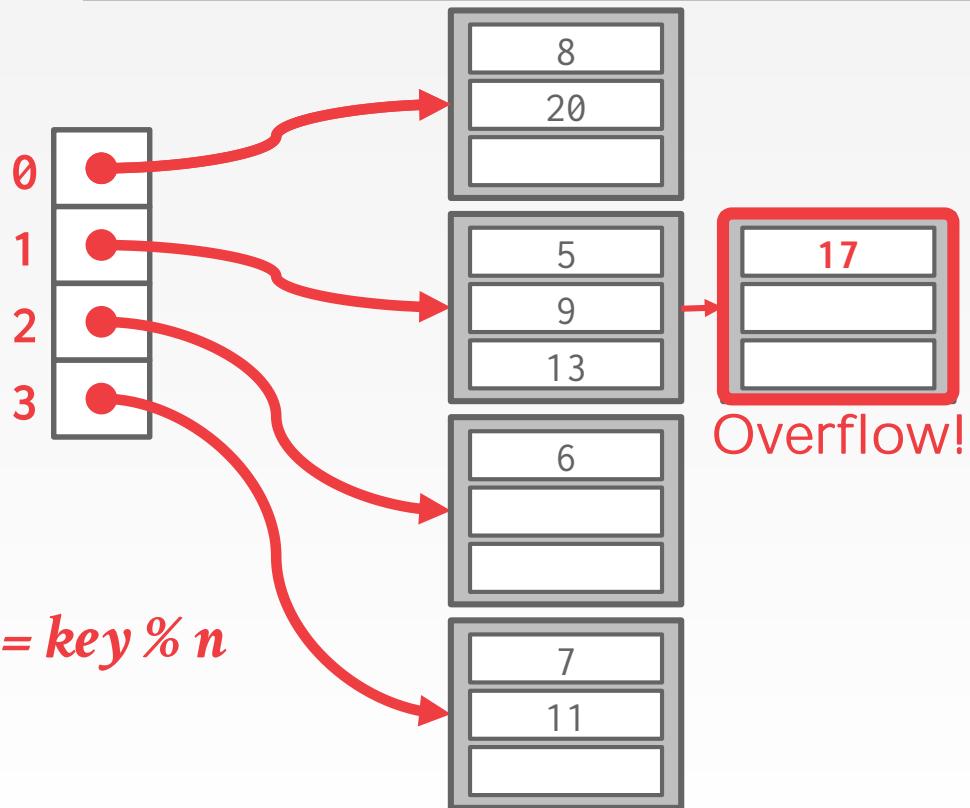
Insert 17

$$hash_1(17) = 17 \% 4 = 1$$



LINEAR HASHING

*Split
Pointer*



Find 6

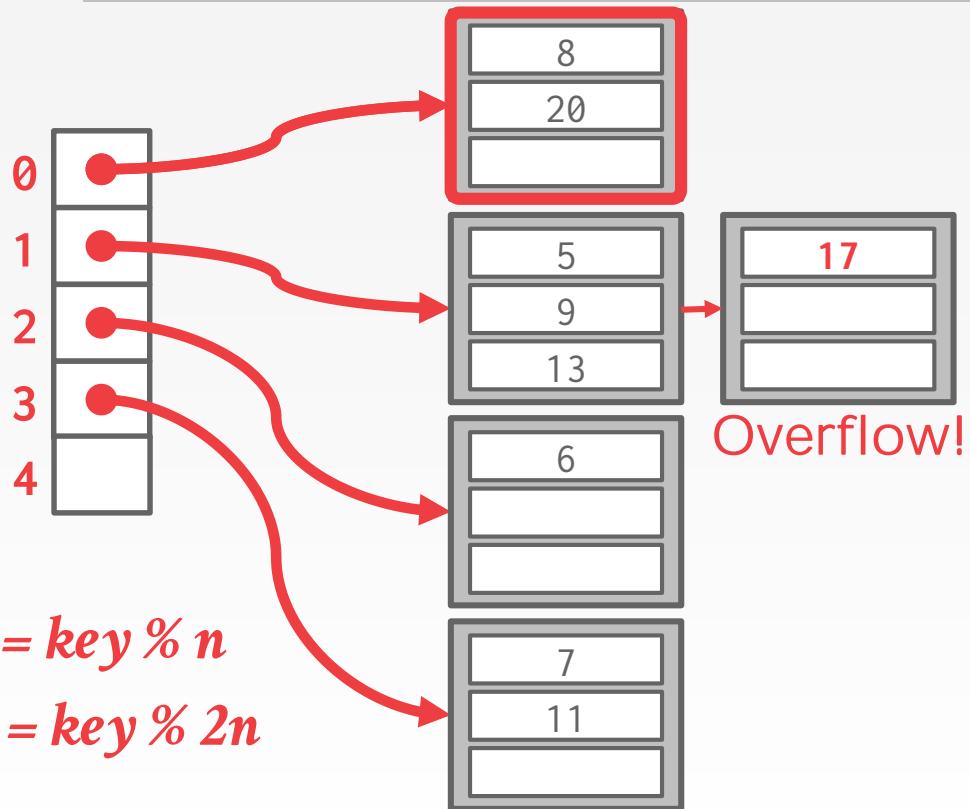
$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

LINEAR HASHING

*Split
Pointer*



$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

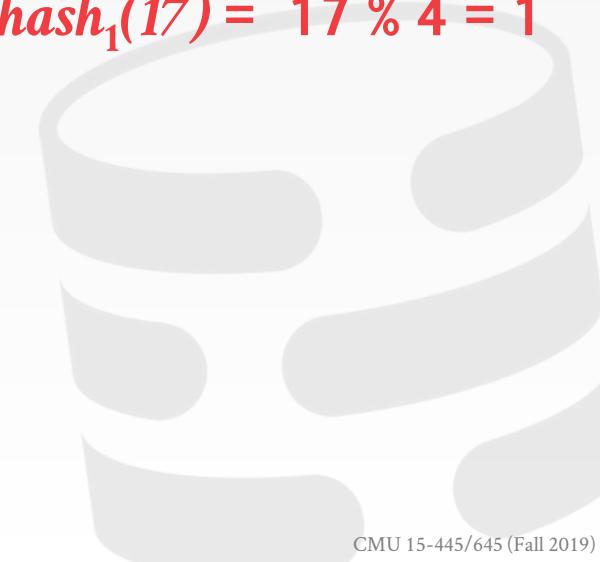
Find 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

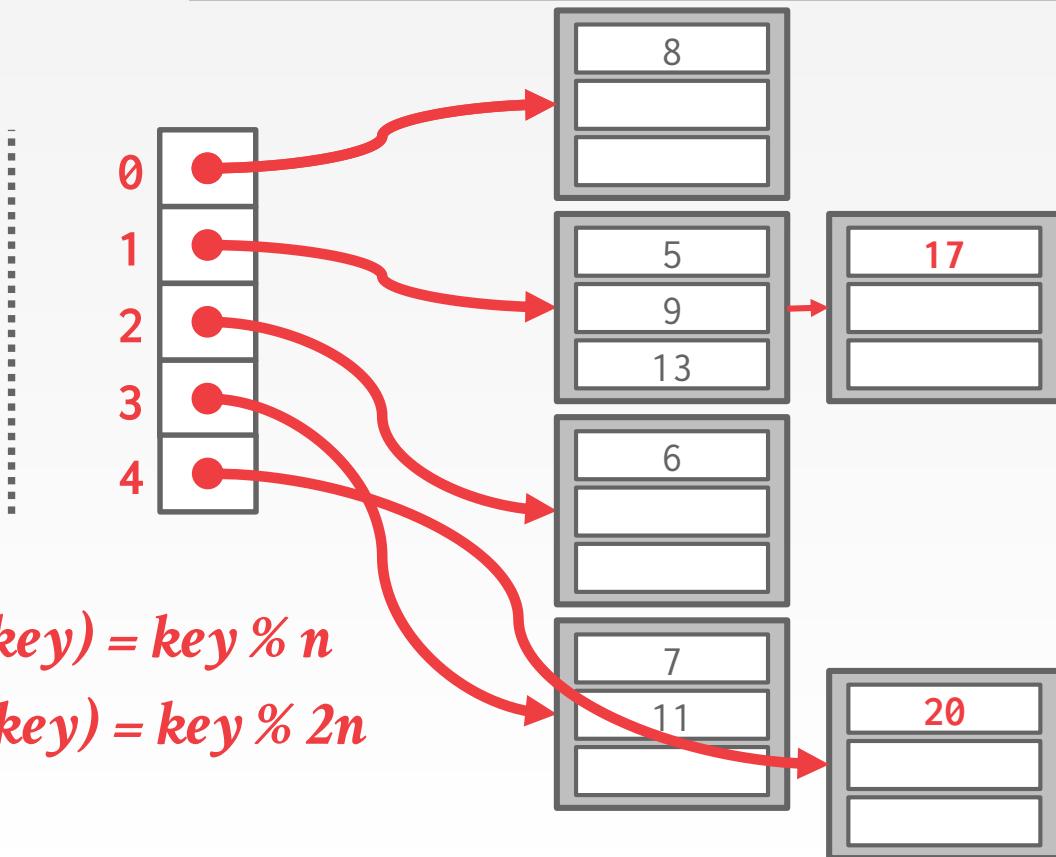
$$\text{hash}_1(17) = 17 \% 4 = 1$$

Overflow!



LINEAR HASHING

*Split
Pointer*



Find 6

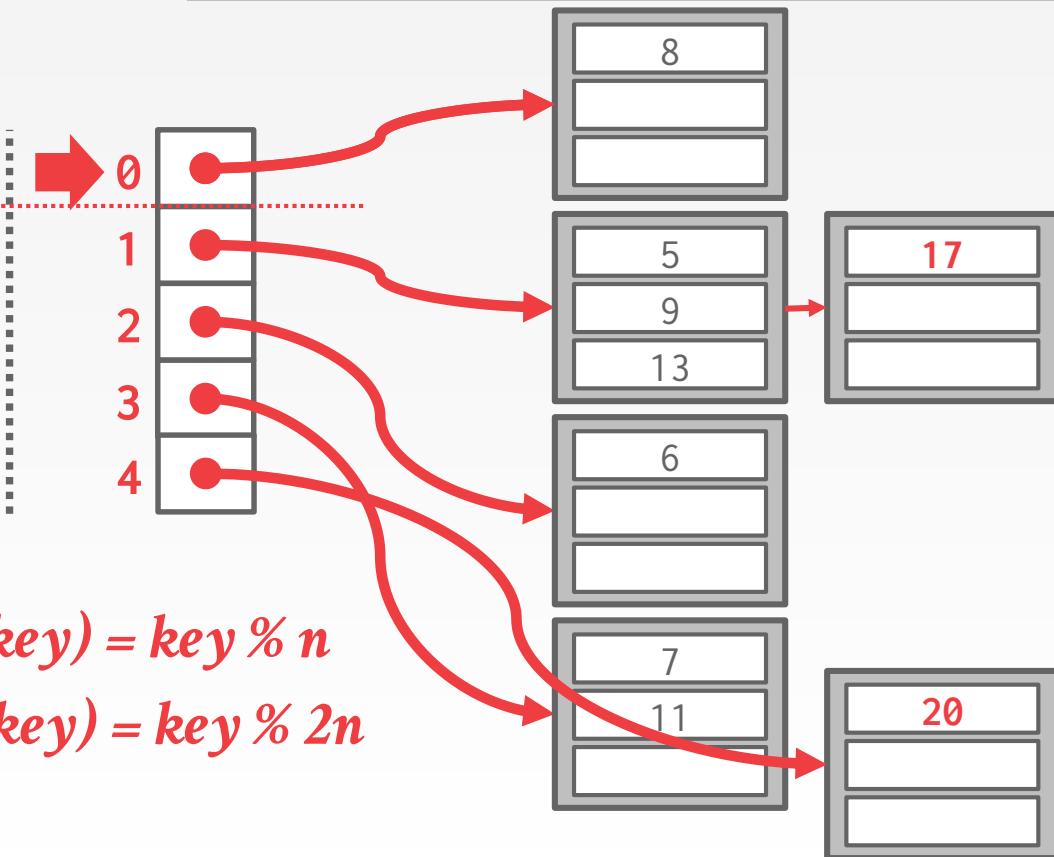
$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

LINEAR HASHING

*Split
Pointer*



$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

Find 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

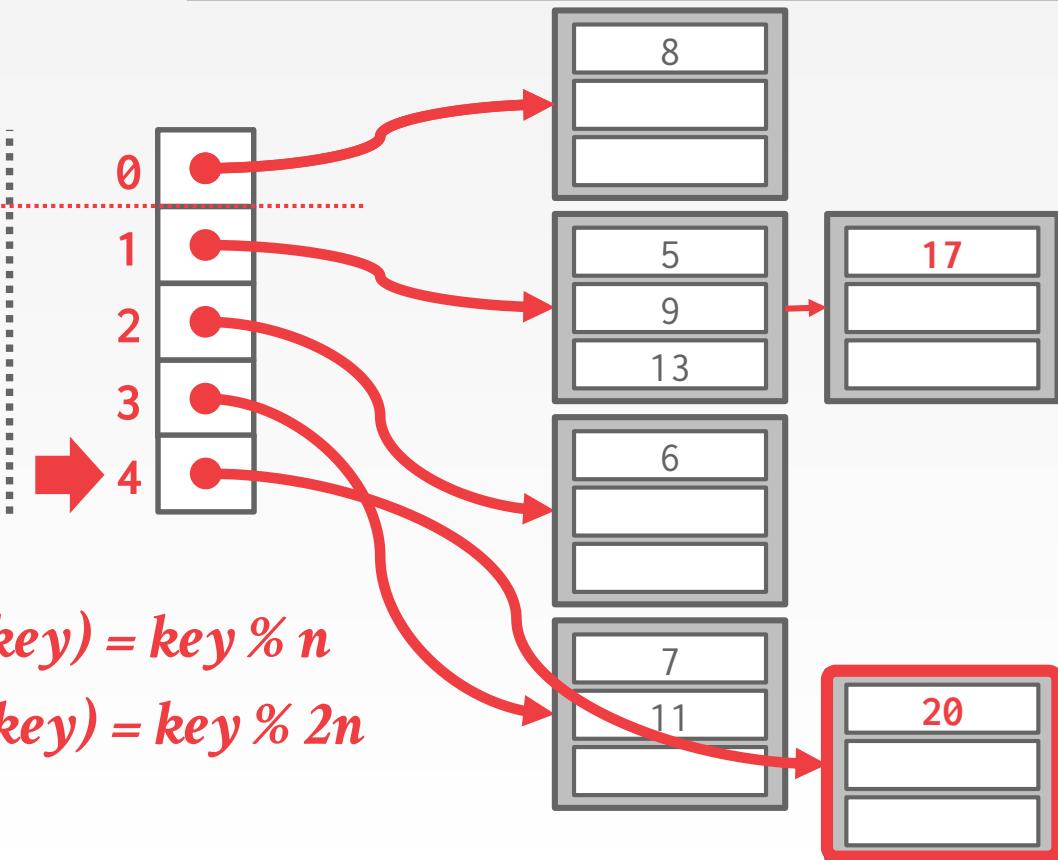
$$\text{hash}_1(17) = 17 \% 4 = 1$$

Find 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

LINEAR HASHING

*Split
Pointer*



Find 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

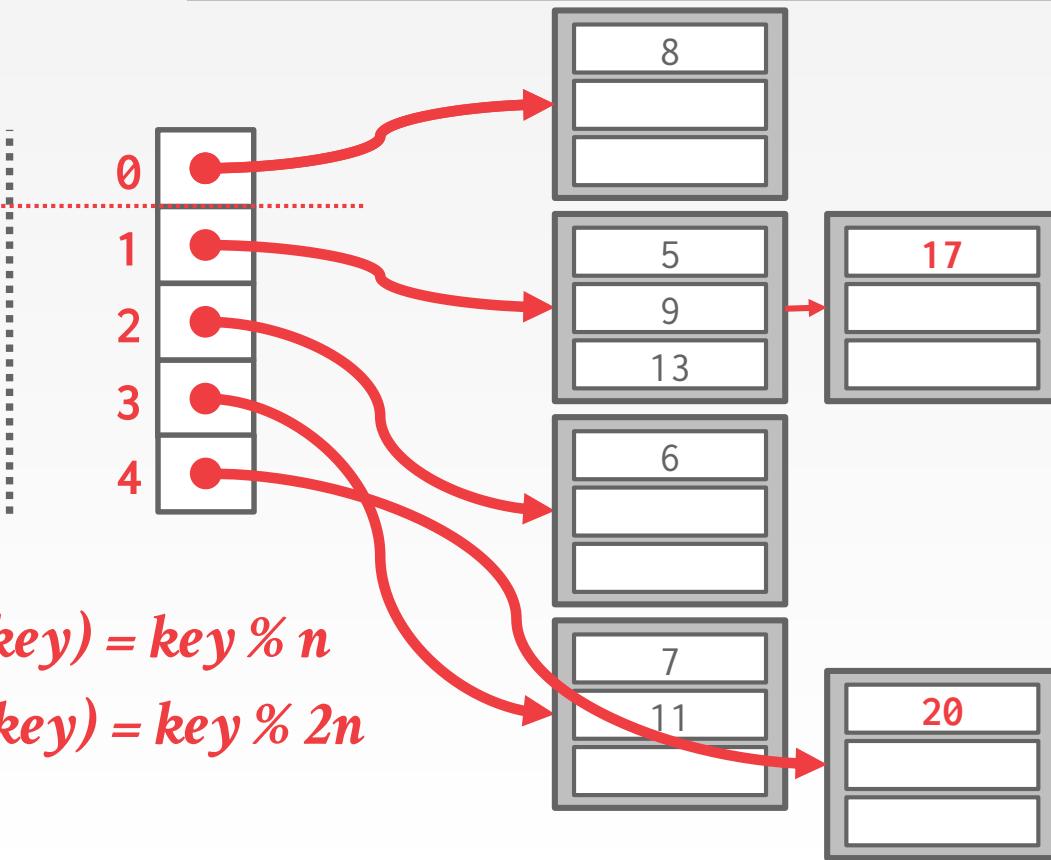
Find 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

LINEAR HASHING

*Split
Pointer*



Find 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

Find 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

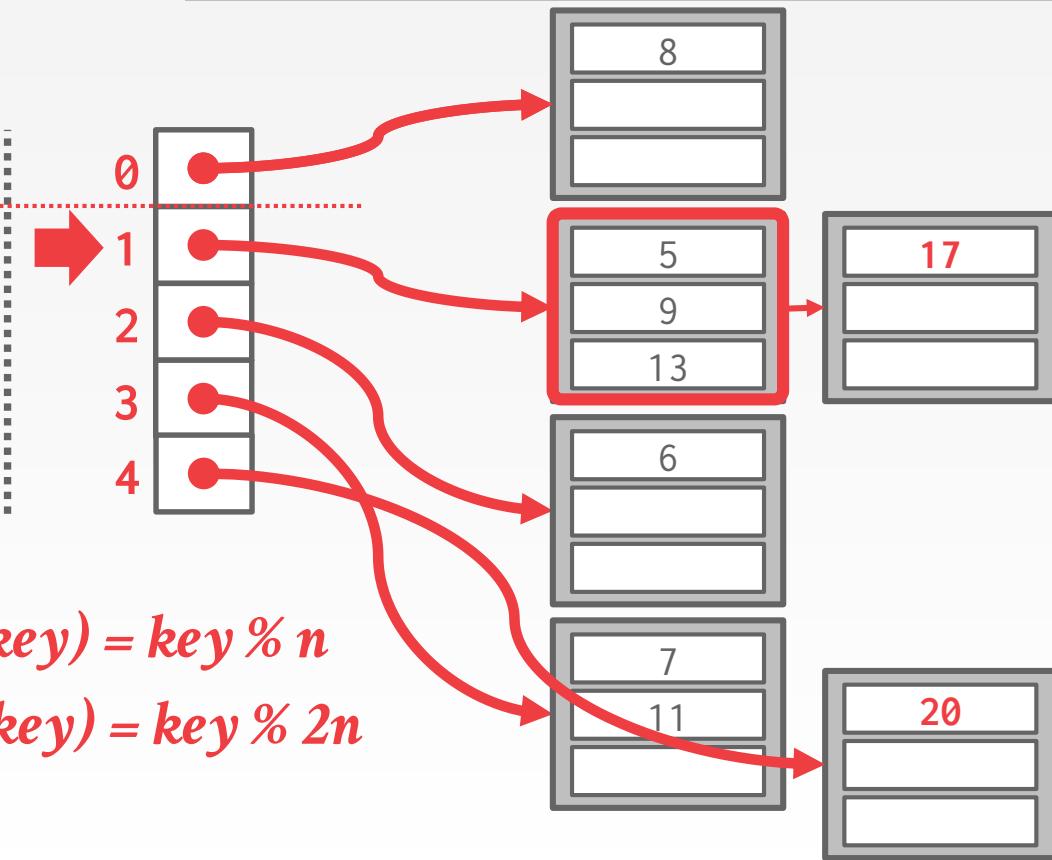
$$\text{hash}_2(20) = 20 \% 8 = 4$$

Find 9

$$\text{hash}_1(9) = 9 \% 4 = 1$$

LINEAR HASHING

*Split
Pointer*



Find 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Insert 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

Find 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

Find 9

$$\text{hash}_1(9) = 9 \% 4 = 1$$

LINEAR HASHING

Splitting buckets based on the split pointer will eventually get to all overflowed buckets.

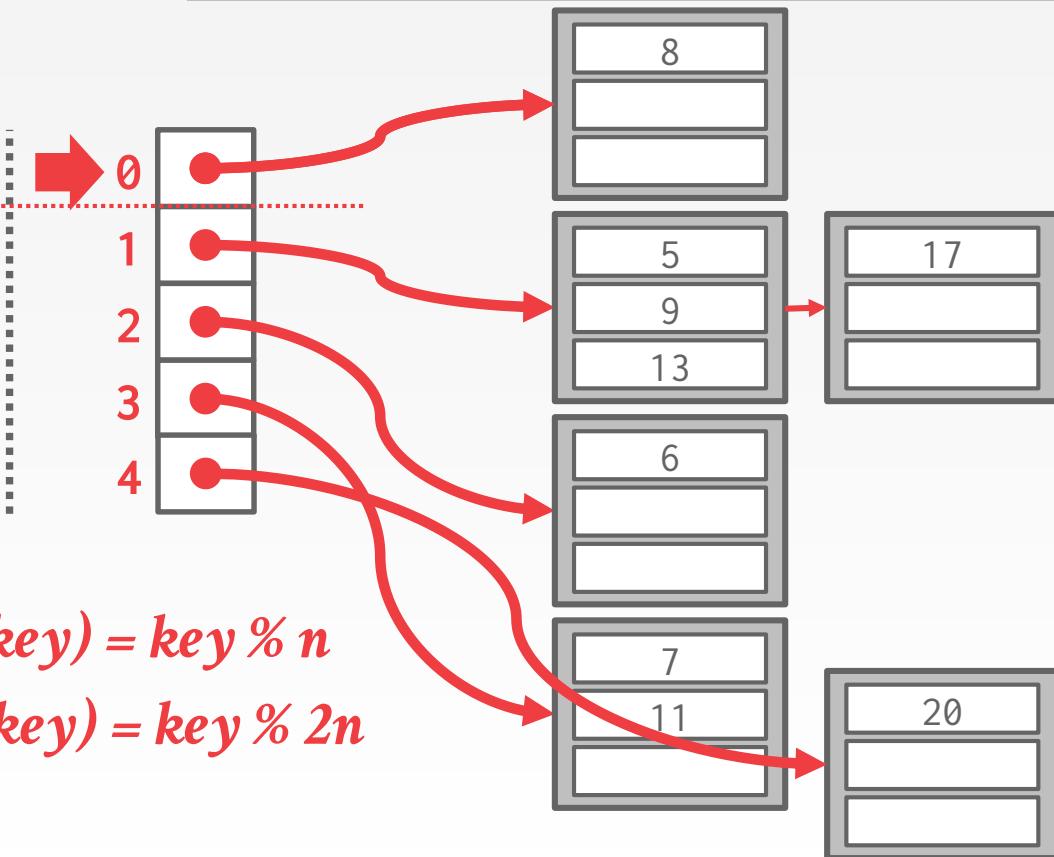
→ When the pointer reaches the last slot, delete the first hash function and move back to beginning.

The pointer can also move backwards when buckets are empty.



LINEAR HASHING – DELETES

*Split
Pointer*



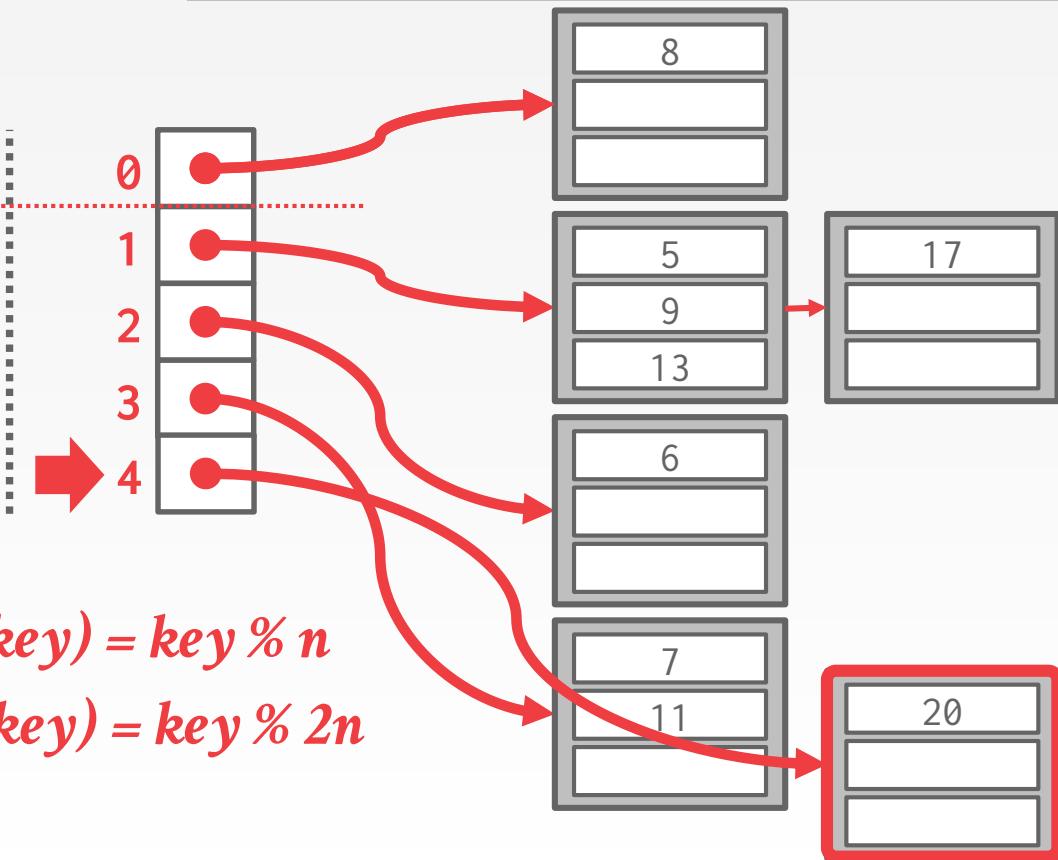
Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$



LINEAR HASHING – DELETES

*Split
Pointer*



Delete 20

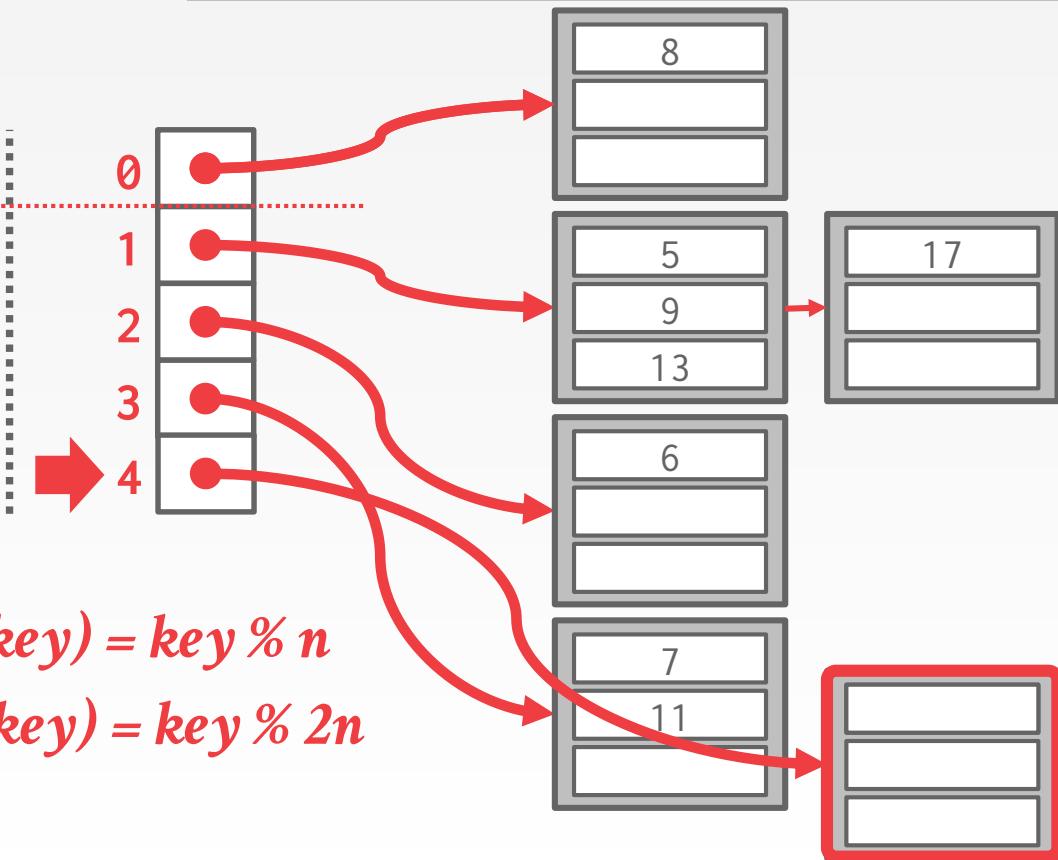
$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$



LINEAR HASHING – DELETES

*Split
Pointer*



$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

Delete 20

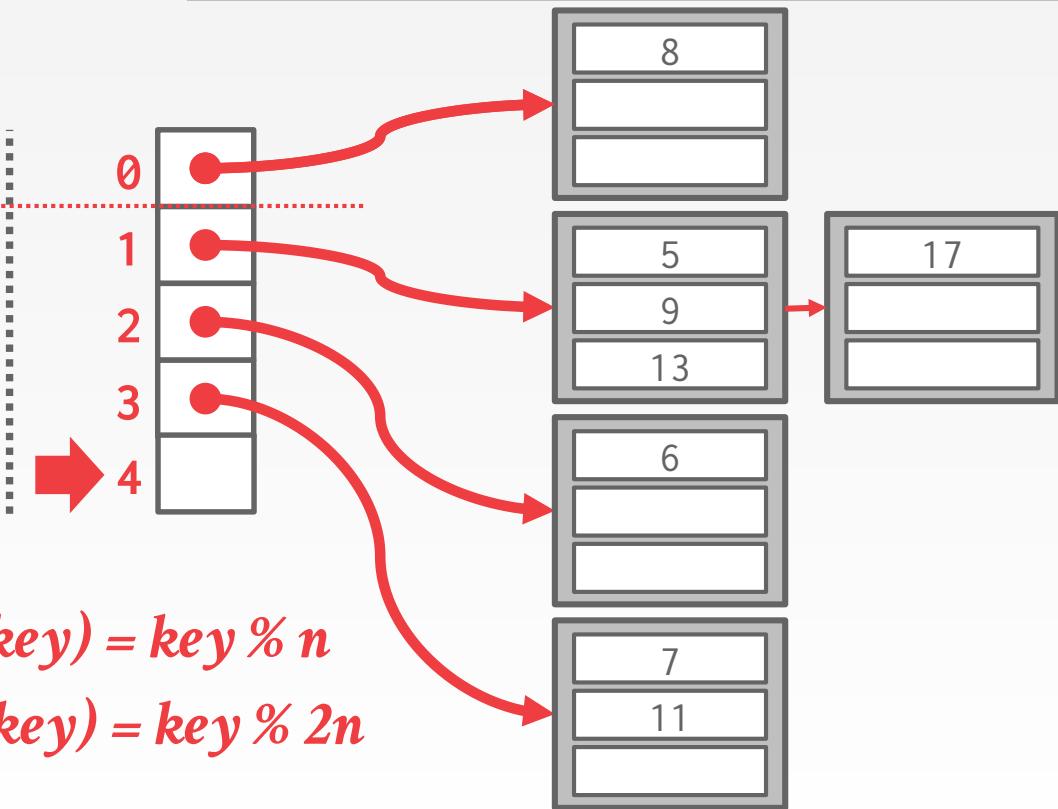
$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$



LINEAR HASHING – DELETES

*Split
Pointer*



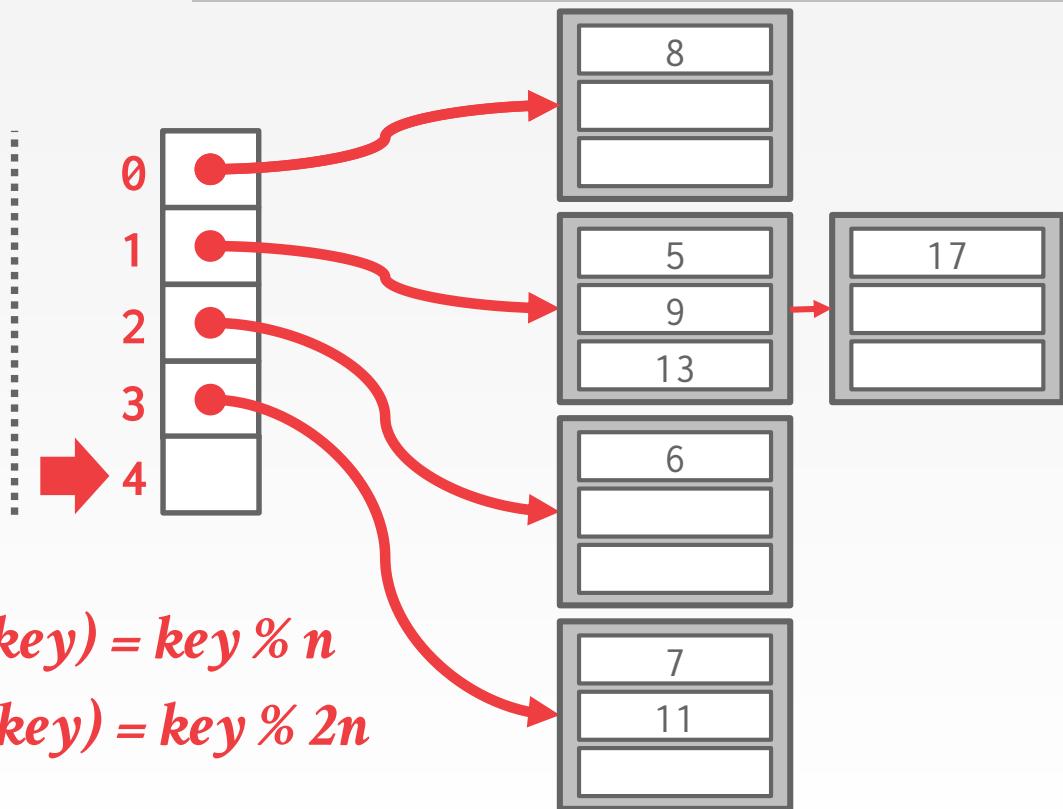
Delete 20

$$\begin{aligned} \text{hash}_1(20) &= 20 \% 4 = 0 \\ \text{hash}_2(20) &= 20 \% 8 = 4 \end{aligned}$$



LINEAR HASHING – DELETES

*Split
Pointer*



Delete 20

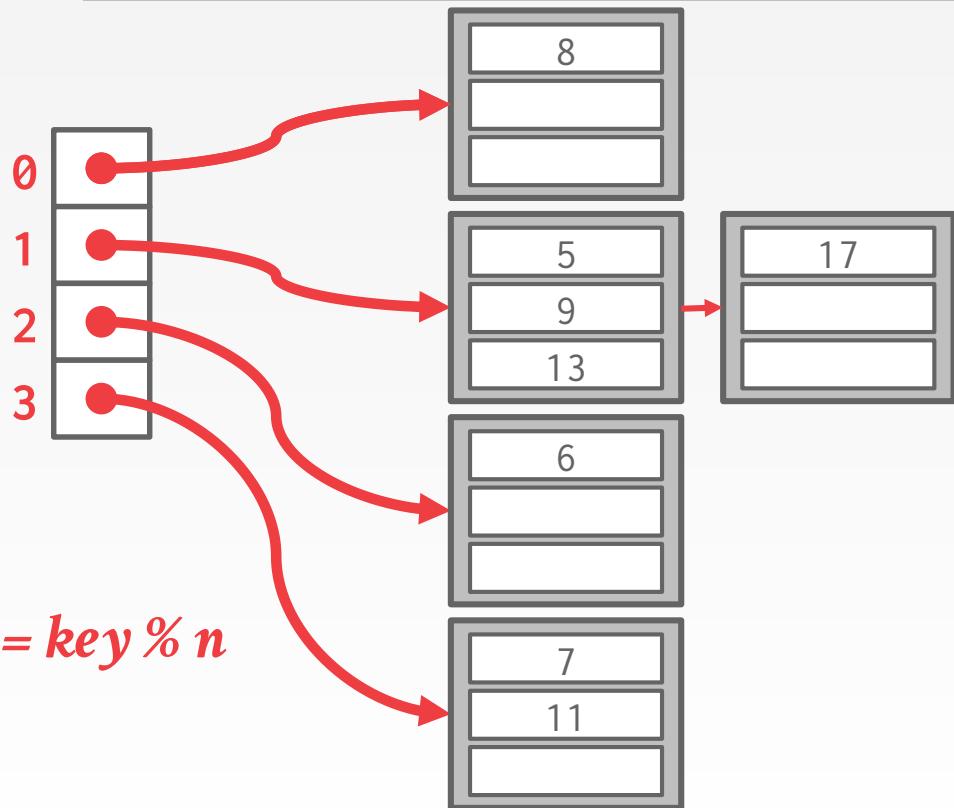
$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$



LINEAR HASHING – DELETES

*Split
Pointer*



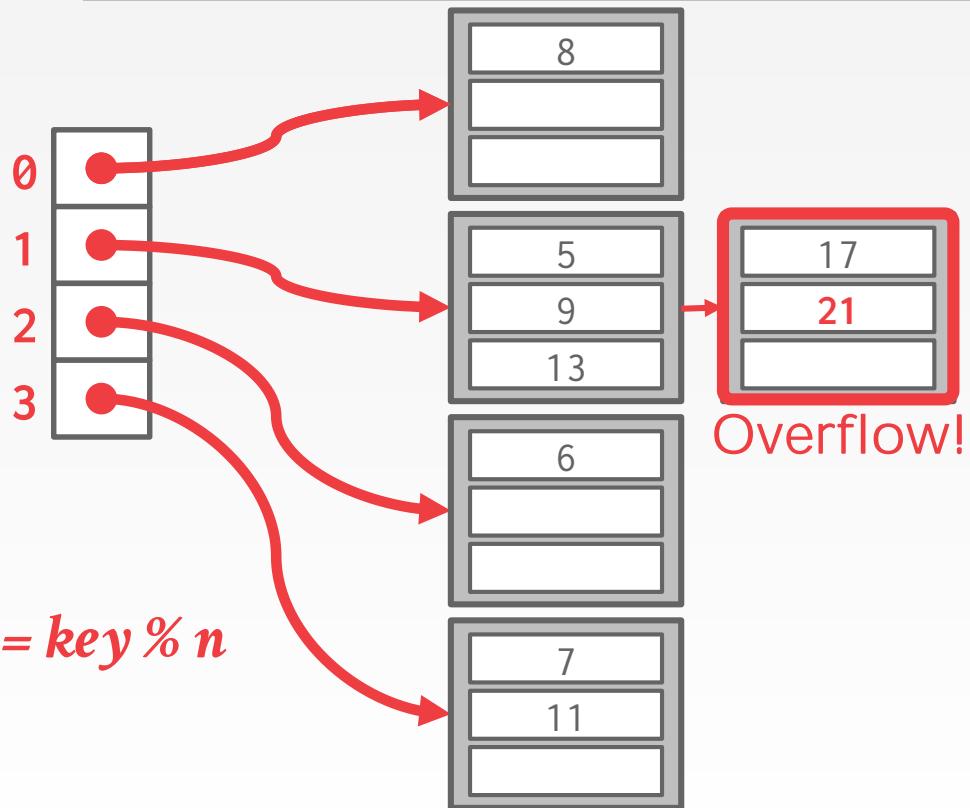
Delete 20

$$\begin{aligned} \text{hash}_1(20) &= 20 \% 4 = 0 \\ \text{hash}_2(20) &= 20 \% 8 = 4 \end{aligned}$$



LINEAR HASHING – DELETES

*Split
Pointer*



Delete 20

$$hash_1(20) = 20 \% 4 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

Insert 21

$$hash_1(21) = 21 \% 4 = 1$$

CONCLUSION

Fast data structures that support **O(1)** look-ups that are used all throughout the DBMS internals.
→ Trade-off between speed and flexibility.

Hash tables are usually not what you want to use for a table index...



NEXT CLASS

B+Trees

→ aka "The Greatest Data Structure of All Time!"



07

Tree Indexes —Part I



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 is due Fri Sept 27th @ 11:59pm

Homework #2 is due Mon Sept 30th @ 11:59pm



DATA STRUCTURES

Internal Meta-data

Core Data Storage

Temporary Data Structures

Table Indexes



TABLE INDEXES

A **table index** is a replica of a subset of a table's attributes that are organized and/or sorted for efficient access using a subset of those attributes.

The DBMS ensures that the contents of the table and the index are logically in sync.



TABLE INDEXES

It is the DBMS's job to figure out the best index(es) to use to execute each query.

There is a trade-off on the number of indexes to create per database.

- Storage Overhead
- Maintenance Overhead



TODAY'S AGENDA

B+Tree Overview

Design Decisions

Optimizations



B-TREE FAMILY

There is a specific data structure called a **B-Tree**.

People also use the term to generally refer to a class of balanced tree data structures:

- **B-Tree** (1971)
- **B+Tree** (1973)
- **B*Tree** (1977?)
- **B^{link}-Tree** (1981)



B-TREE FAMILY

There is a specific data structure called

People also use the term to generally refer to a class of balanced tree data structures.

- **B-Tree** (1971)
- **B+Tree** (1973)
- **B*Trees** (1977?)
- **B^{link}-Tree** (1981)

Efficient Locking for Concurrent Operations on B-Trees

PHILIP L. LEHMAN
Carnegie-Mellon University
and
S. BING YAO
Purdue University

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given.

Keywords and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees
CR Categories: *3.7.3, 3.7.4, 4.3.2, 4.3.3, 4.34, 5.24*

1. INTRODUCTION

The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

A topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this paper, we consider a simple variant of the B-tree (actually of the B⁺-tree, system).

Methods for concurrent operations on B^{*}-trees have been discussed by Bayer and Schkolnick [3] and others [6, 12, 13]. The solution given in the current paper

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported by the National Science Foundation under Grant MCS76-16604. Authors' present addresses: P. L. Lehman, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; S. B. Yao, Department of Computer Science and College of Business and Management, University of Maryland, College Park, MD 20742.
 © 1981 ACM 0892-3915/81/1200-0650 \$00.75

ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, Pages 650-670.

B + TREE

A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in **O(log n)**.

- Generalization of a binary search tree in that a node can have more than two children.
- Optimized for systems that read and write large blocks of data.

The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees. This paper reviews B-trees and shows why they have been so successful. It discusses the major variations of the B-tree, especially the B*-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B'-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4.34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names, addresses, and salaries." and
Random: "From our employee file, extract the information about employee J. Smith".

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A-G," "H-R," and "S-Z," while the folders

might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. Figure 1 depicts a file and its index. An index may be associated with a file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost level consists of labels on drawers, and the next level of index consists of labels on folders.

Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best performance. To copy without fee all or part of this material is granted provided that the copies are not made or distributed for commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

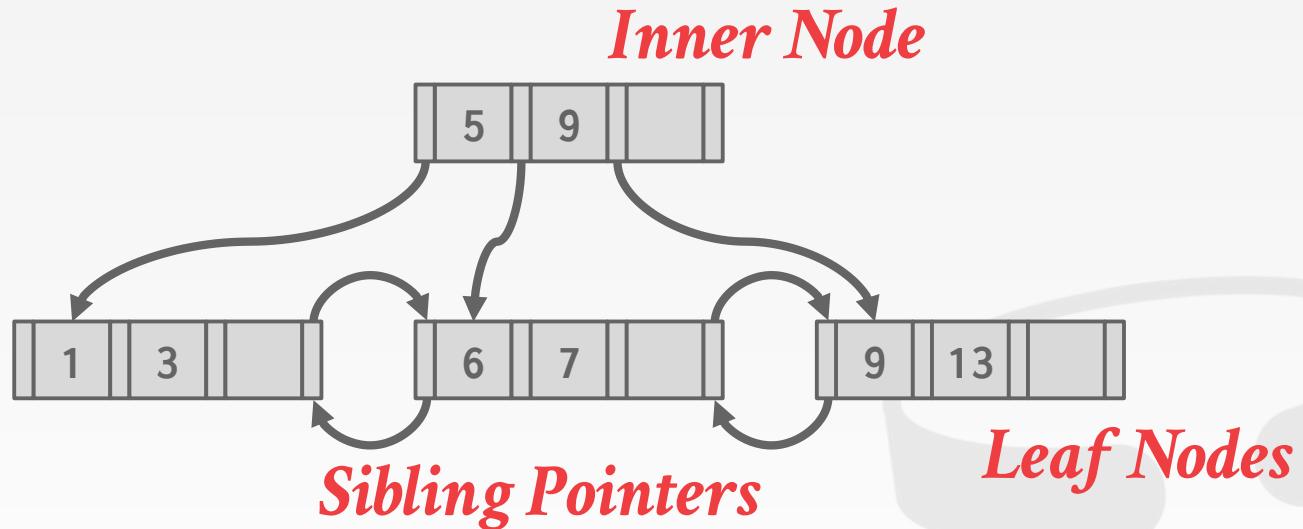
© 1979 ACM 0010-4892/79/0600-0121 \$0.75
Computing Surveys, Vol. 11, No. 2, June 1979

B+TREE PROPERTIES

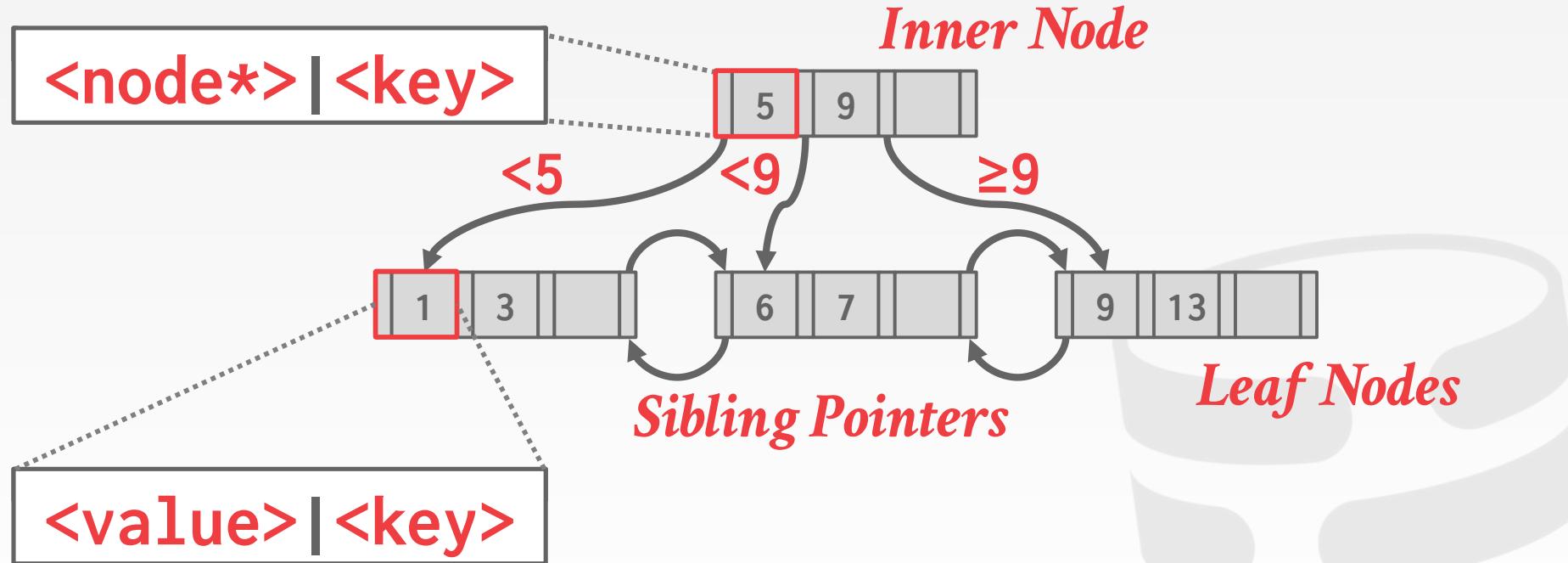
A B+Tree is an M -way search tree with the following properties:

- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every node other than the root, is at least half-full
 $M/2-1 \leq \#keys \leq M-1$
- Every inner node with k keys has $k+1$ non-null children

B + TREE EXAMPLE



B + TREE EXAMPLE



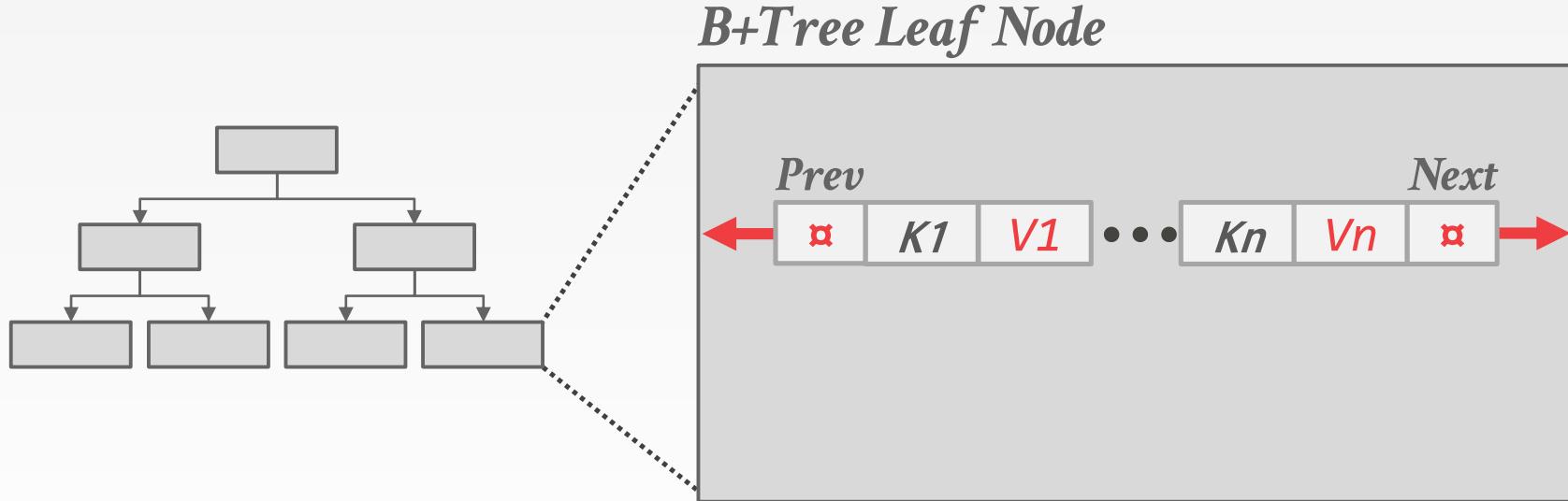
NODES

Every B+Tree node is comprised of an array of key/value pairs.

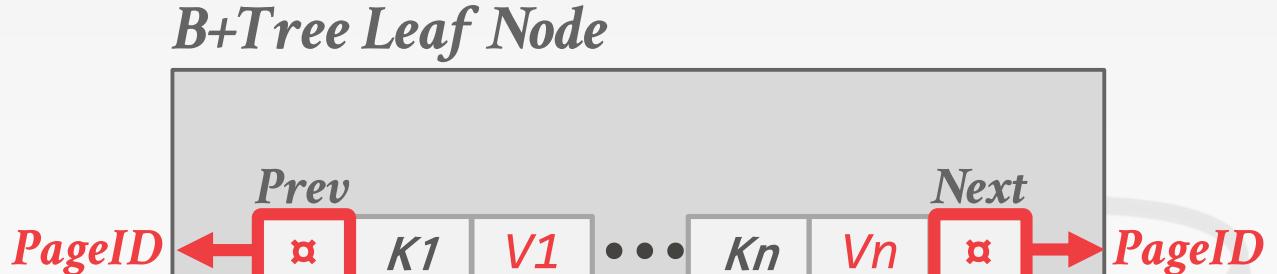
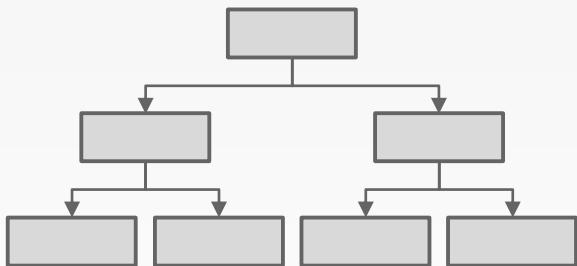
- The keys are derived from the attributes(s) that the index is based on.
- The values will differ based on whether the node is classified as **inner nodes** or **leaf nodes**.

The arrays are (usually) kept in sorted key order.

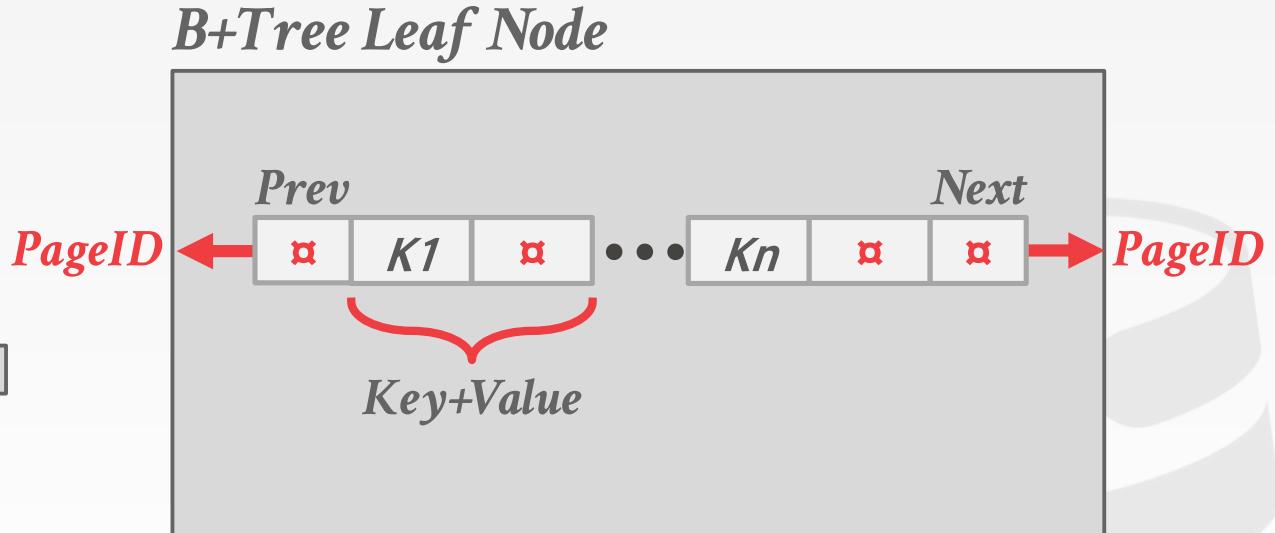
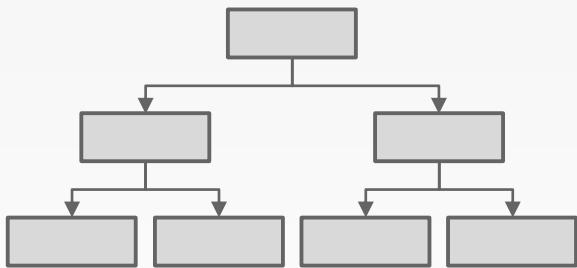
B+TREE LEAF NODES



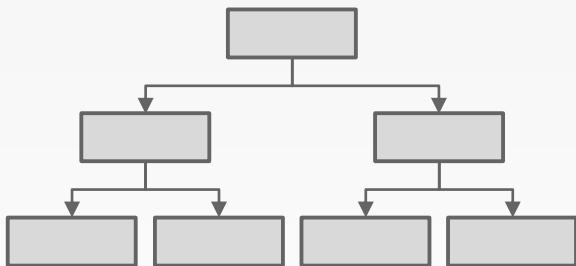
B+TREE LEAF NODES



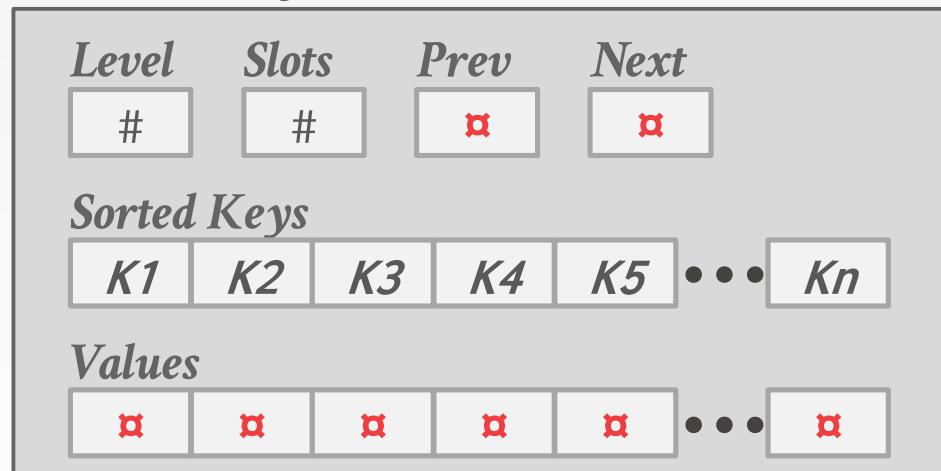
B+TREE LEAF NODES



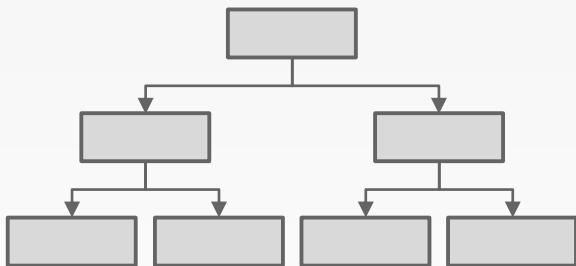
B+TREE LEAF NODES



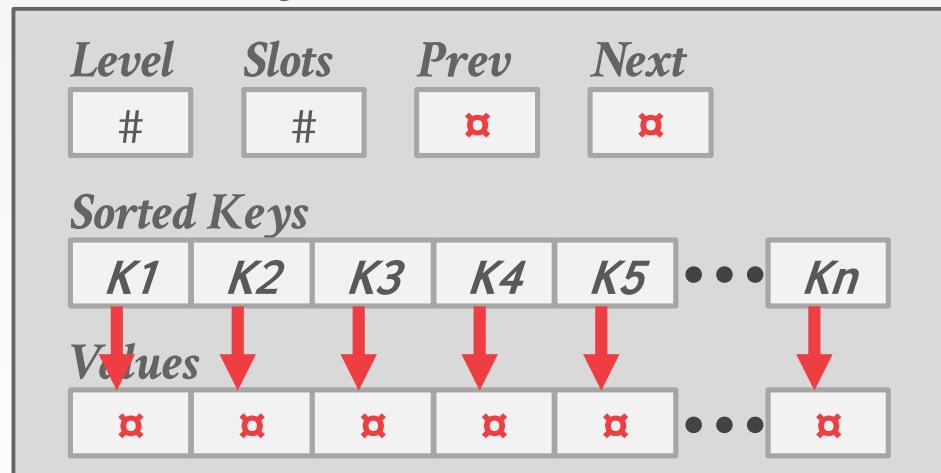
B+Tree Leaf Node



B+TREE LEAF NODES



B+Tree Leaf Node



LEAF NODE VALUES

Approach #1: Record Ids

- A pointer to the location of the tuple that the index entry corresponds to.

Approach #2: Tuple Data

- The actual contents of the tuple is stored in the leaf node.
- Secondary indexes have to store the record id as their values.



B-TREE VS. B+TREE

The original **B-Tree** from 1972 stored keys + values in all nodes in the tree.

→ More space efficient since each key only appears once in the tree.

A **B+Tree** only stores values in leaf nodes. Inner nodes only guide the search process.

B+TREE INSERT

Find correct leaf node **L**.

Put data entry into **L** in sorted order.

If **L** has enough space, done!

Otherwise, split **L** keys into **L** and a new node **L2**

- Redistribute entries evenly, copy up middle key.

- Insert index entry pointing to **L2** into parent of **L**.

To split inner node, redistribute entries evenly,
but push up middle key.

B+TREE VISUALIZATION

<https://cmudb.io/btree>

Source: [David Gales \(Univ. of San Francisco\)](#)



B+TREE DELETE

Start at root, find leaf **L** where entry belongs.

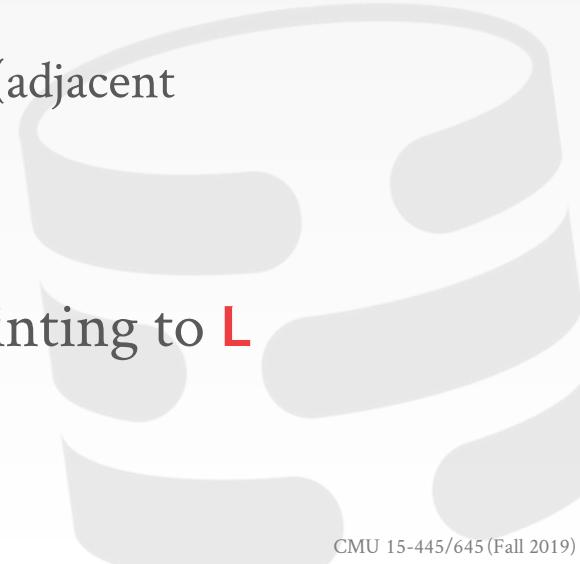
Remove the entry.

If **L** is at least half-full, done!

If **L** has only **M/2-1** entries,

- Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).
- If re-distribution fails, merge **L** and sibling.

If merge occurred, must delete entry (pointing to **L** or sibling) from parent of **L**.



B+TREES IN PRACTICE

Typical Fill-Factor: 67%.

Typical Capacities:

- Height 4: $1334 = 312,900,721$ entries
- Height 3: $1333 = 2,406,104$ entries

Pages per level:

- Level 1 = 1 page = 8 KB
- Level 2 = 134 pages = 1 MB
- Level 3 = 17,956 pages = 140 MB



CLUSTERED INDEXES

The table is stored in the sort order specified by the primary key.

→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.

→ If a table doesn't contain a primary key, the DBMS will automatically make a hidden row id primary key.

Other DBMSs cannot use them at all.

SELECTION CONDITIONS

The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

Example: Index on **<a , b , c>**

- Supported: **(a=5 AND b=3)**
- Supported: **(b=3)**.

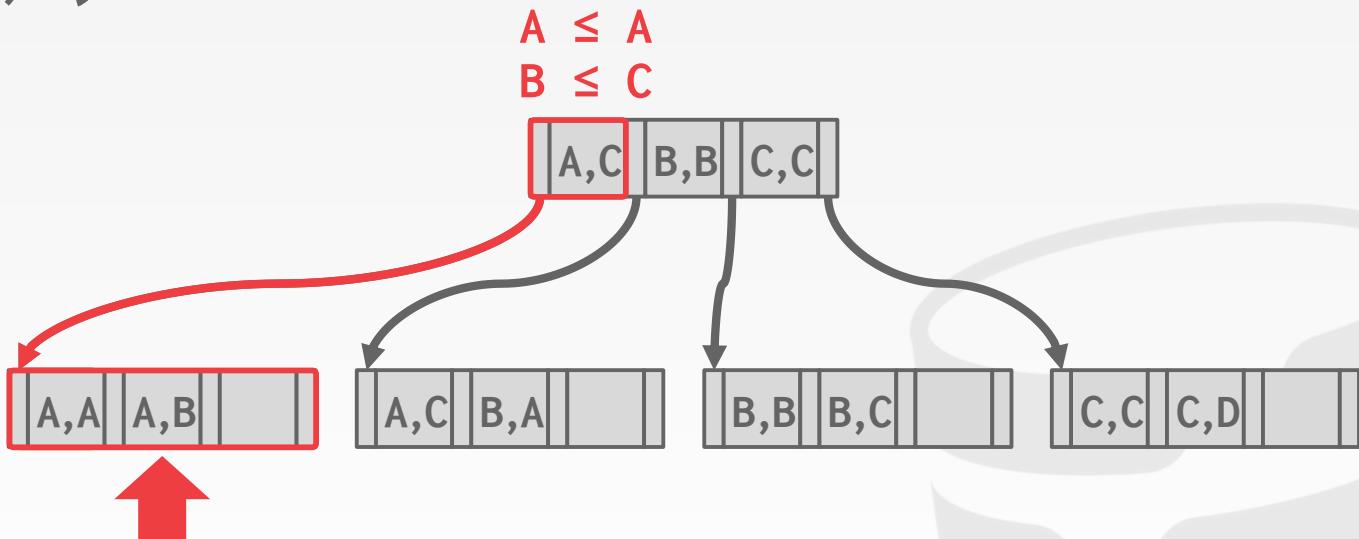
Not all DBMSs support this.

For hash index, we must have all attributes in search key.



SELECTION CONDITIONS

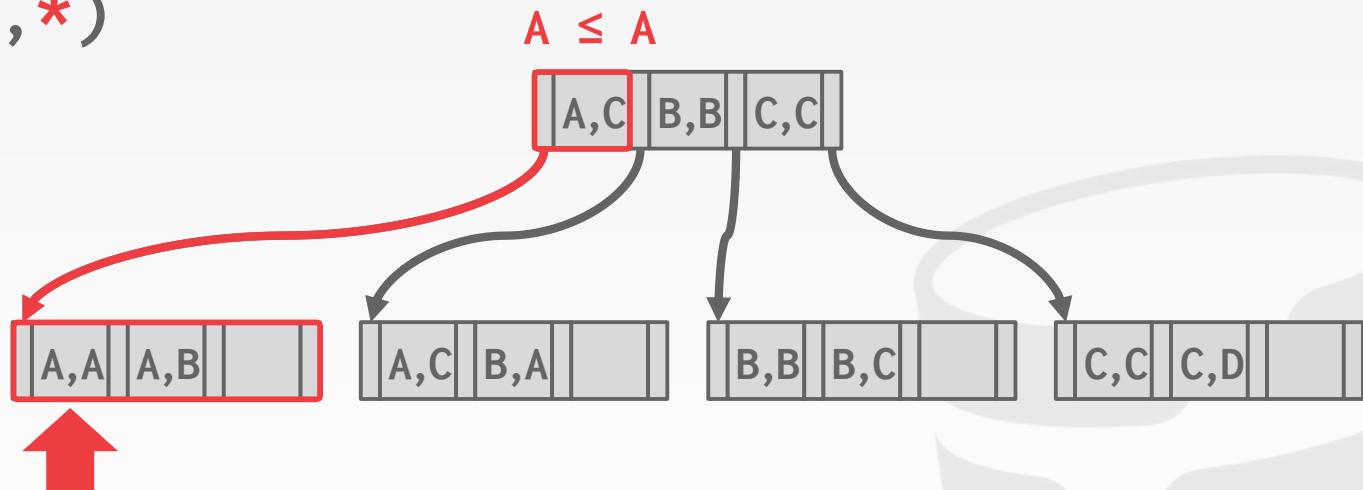
Find Key=(A,B)



SELECTION CONDITIONS

Find Key=(A,B)

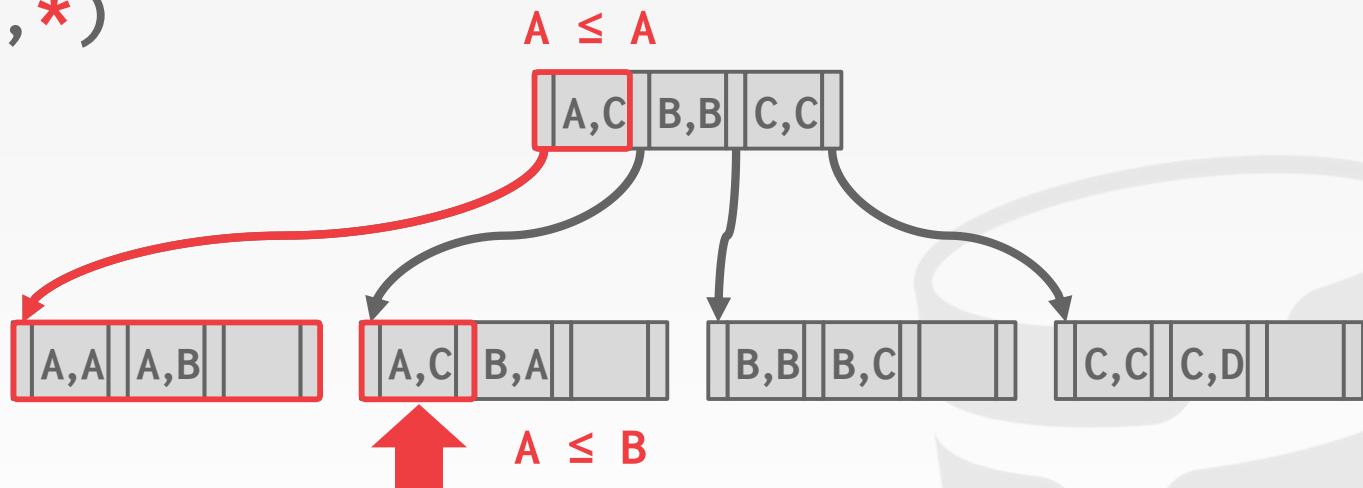
Find Key=(A,*)



SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

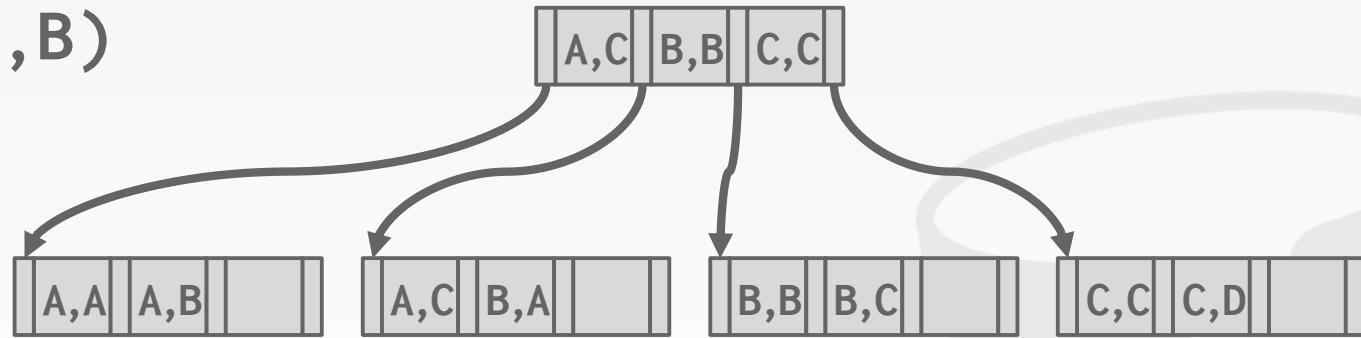


SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A, *)

Find Key=(*, B)

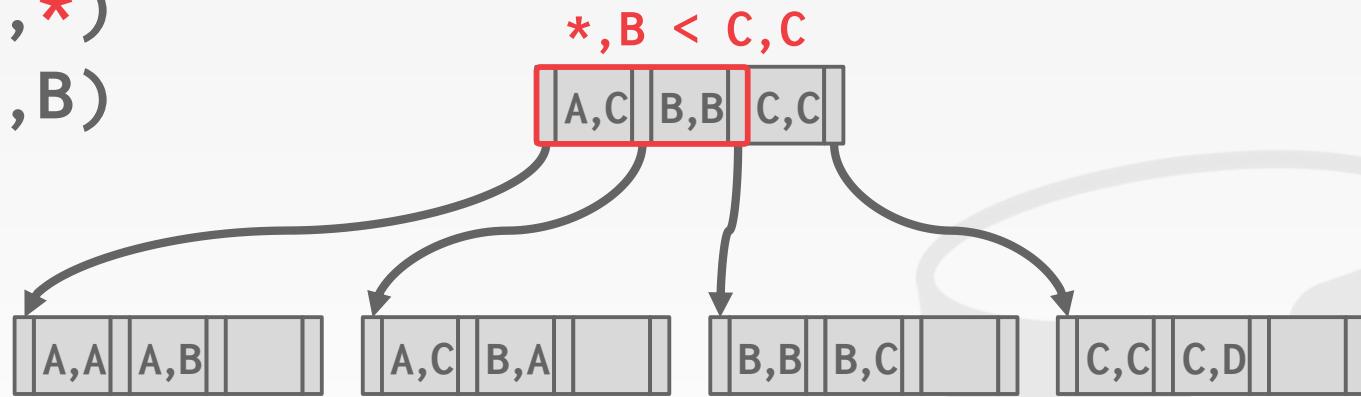


SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A, *)

Find Key=(*, B)

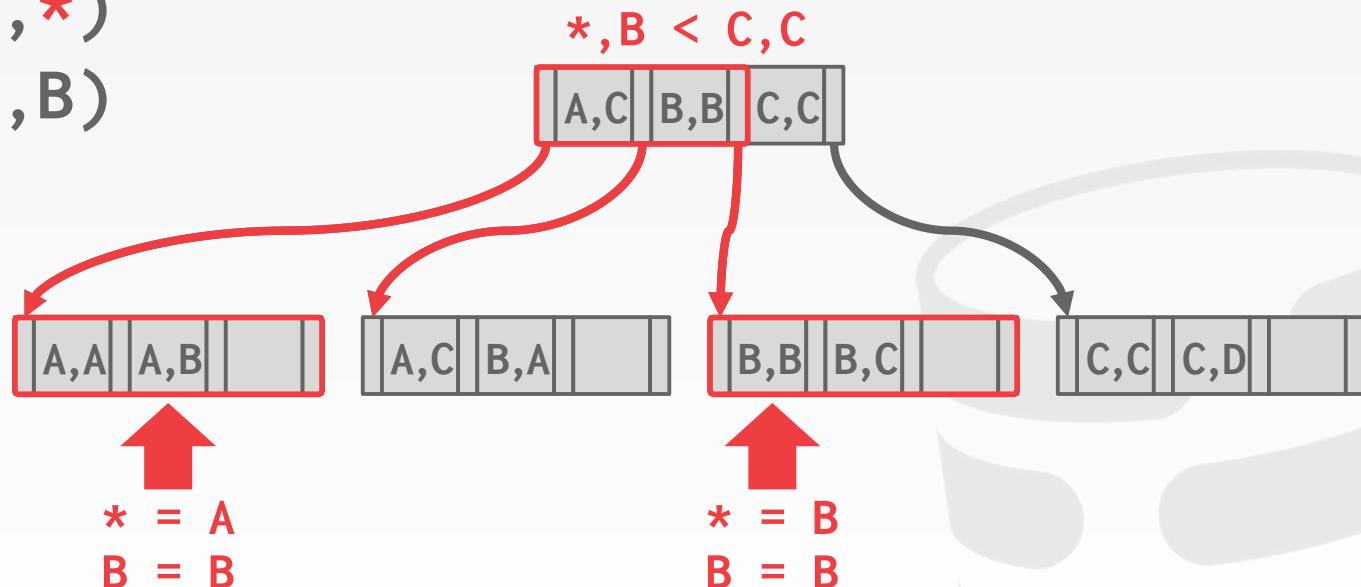


SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)

Find Key=(*,B)



B+TREE DESIGN CHOICES

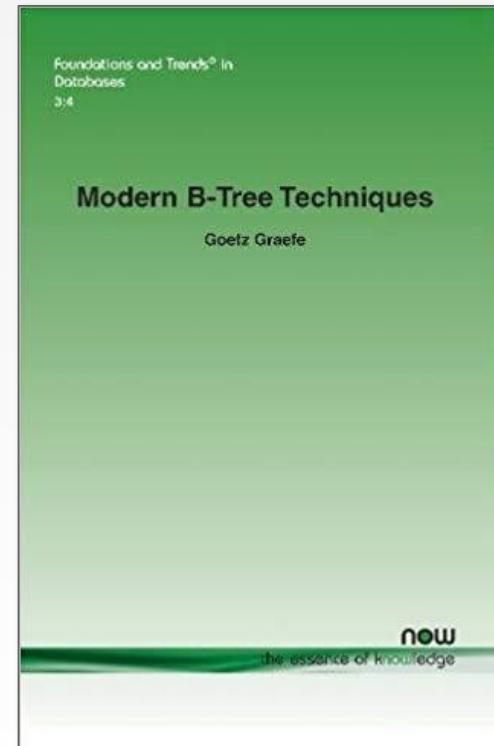
Node Size

Merge Threshold

Variable Length Keys

Non-Unique Indexes

Intra-Node Search



NODE SIZE

The slower the storage device, the larger the optimal node size for a B+Tree.

- HDD ~1MB
- SSD: ~10KB
- In-Memory: ~512B

Optimal sizes can vary depending on the workload
→ Leaf Node Scans vs. Root-to-Leaf Traversals

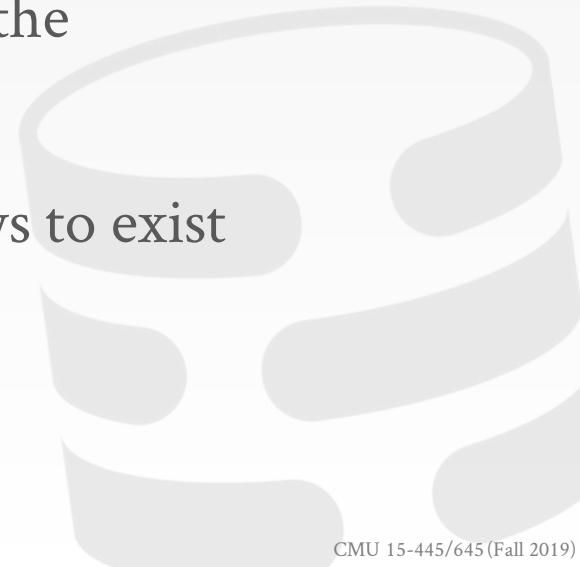


MERGE THRESHOLD

Some DBMSs do not always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.

It may also be better to just let underflows to exist and then periodically **rebuild** entire tree.



VARIABLE LENGTH KEYS

Approach #1: Pointers

- Store the keys as pointers to the tuple's attribute.

Approach #2: Variable Length Nodes

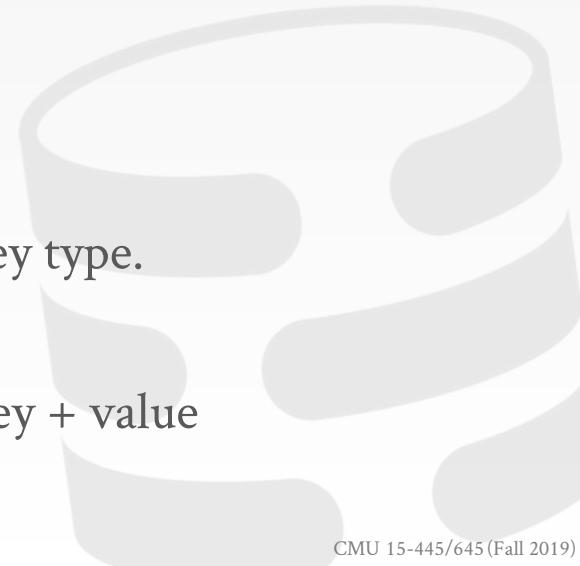
- The size of each node in the index can vary.
- Requires careful memory management.

Approach #3: Padding

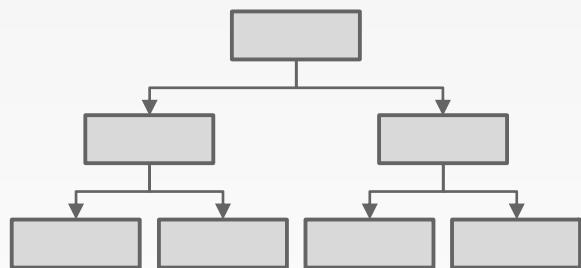
- Always pad the key to be max length of the key type.

Approach #4: Key Map / Indirection

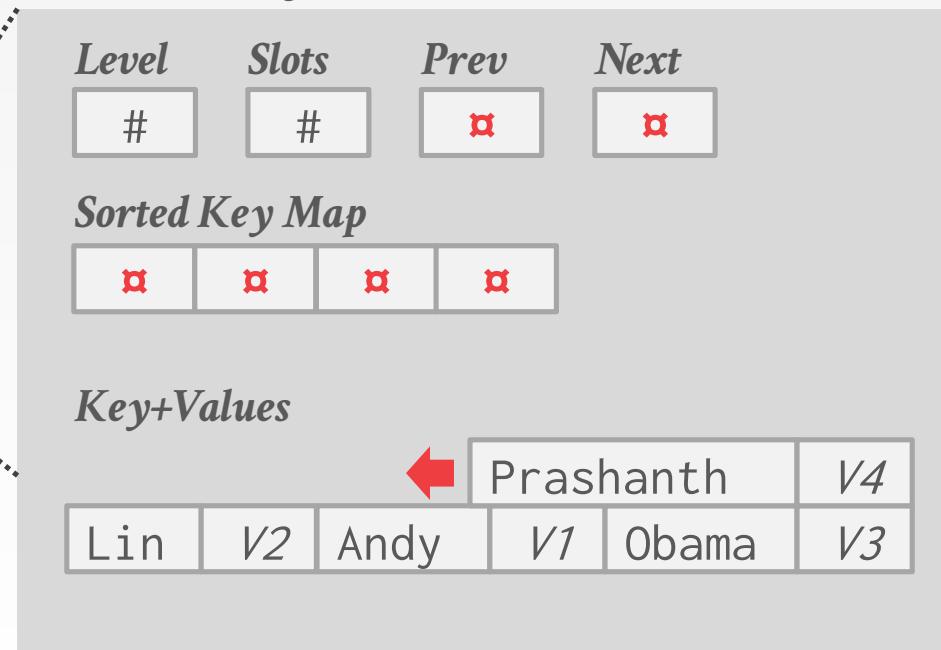
- Embed an array of pointers that map to the key + value list within the node.



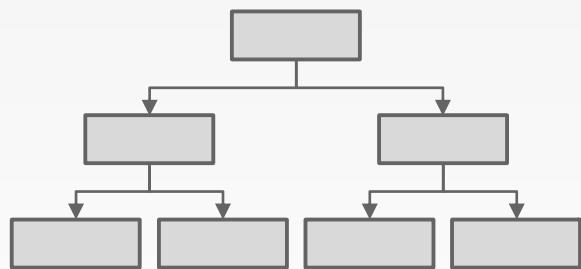
KEY MAP / INDIRECTION



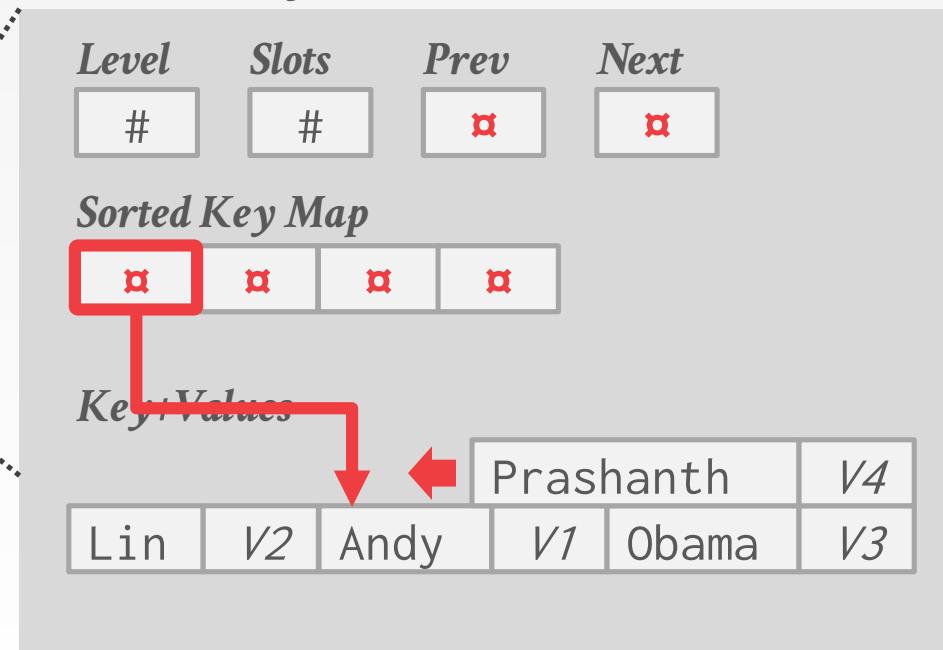
B+Tree Leaf Node



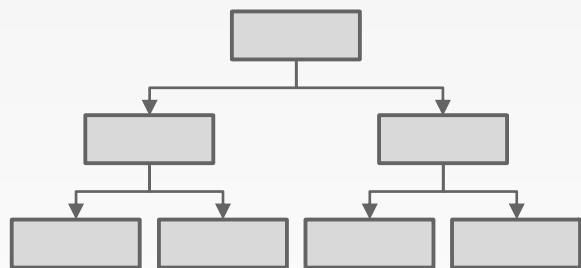
KEY MAP / INDIRECTION



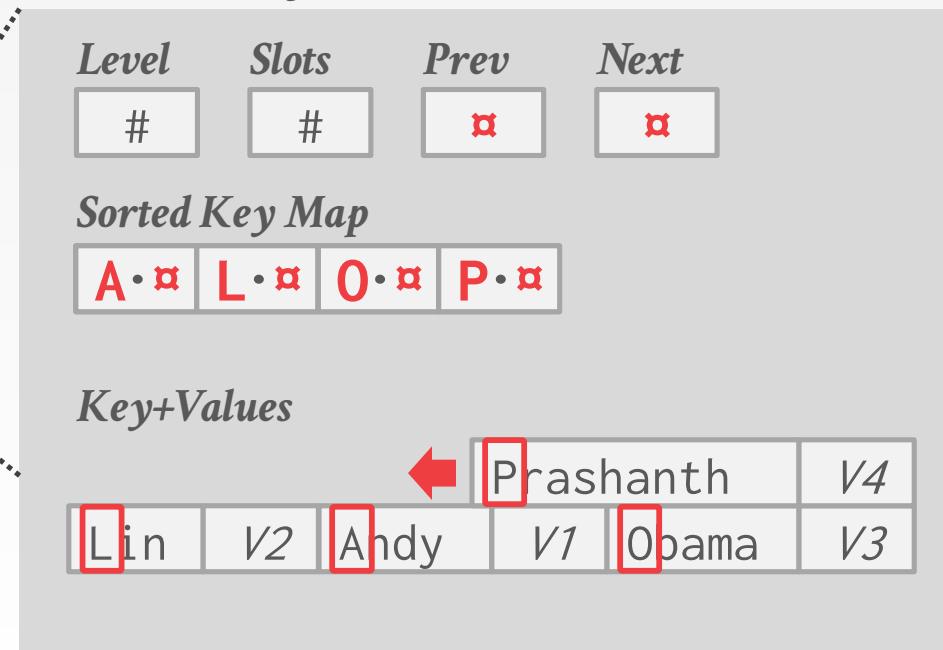
B+Tree Leaf Node



KEY MAP / INDIRECTION



B+Tree Leaf Node



NON-UNIQUE INDEXES

Approach #1: Duplicate Keys

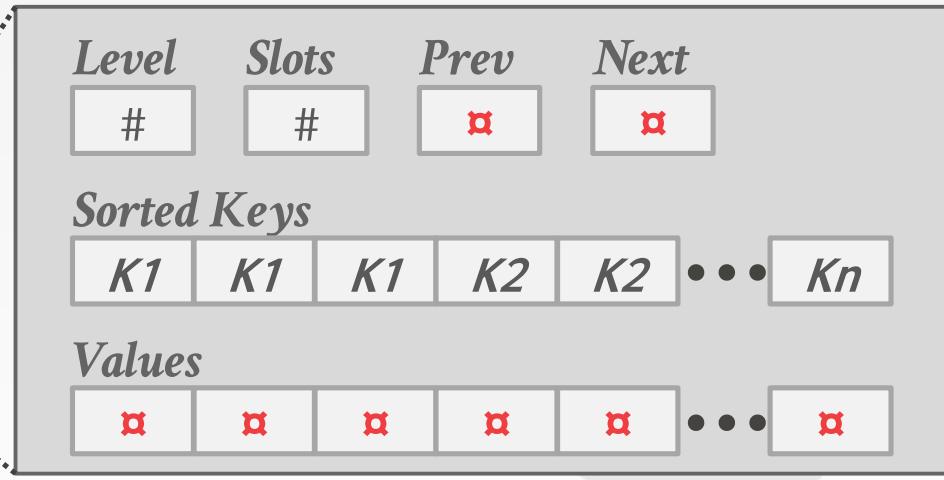
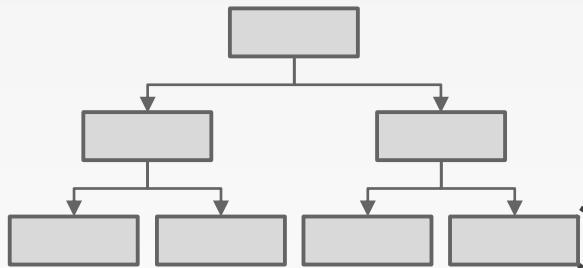
- Use the same leaf node layout but store duplicate keys multiple times.

Approach #2: Value Lists

- Store each key only once and maintain a linked list of unique values.

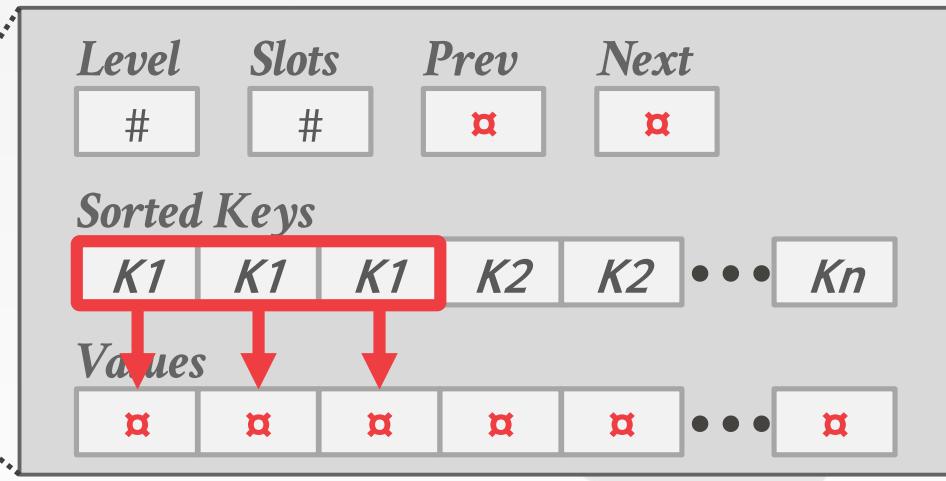
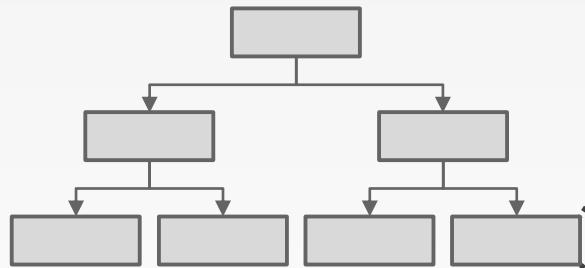
NON-UNIQUE: DUPLICATE KEYS

B+Tree Leaf Node



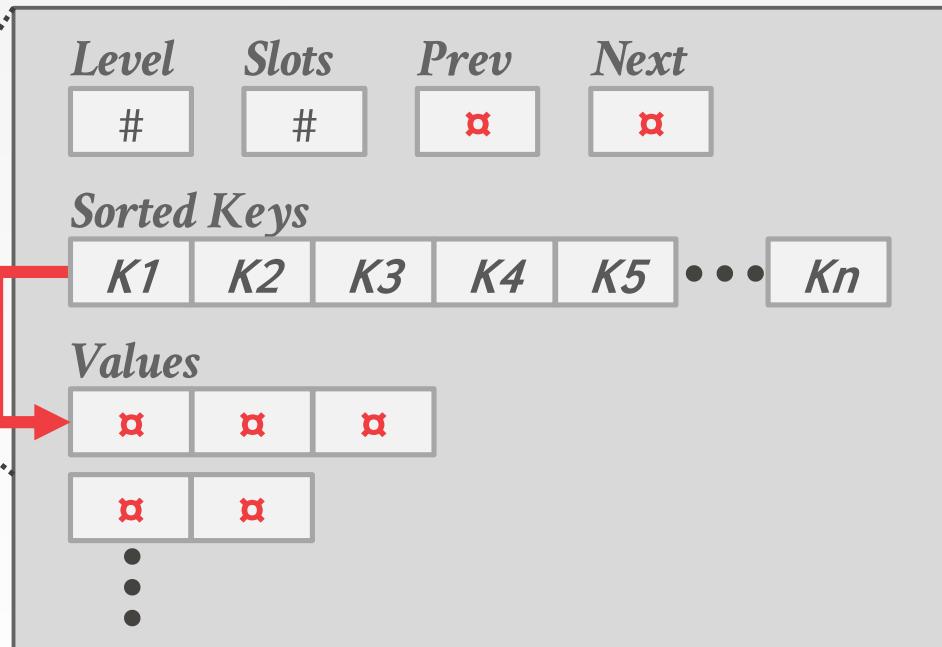
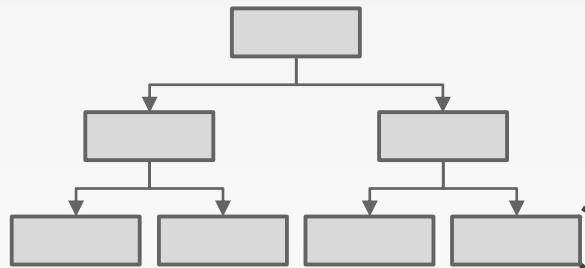
NON-UNIQUE: DUPLICATE KEYS

B+Tree Leaf Node



NON-UNIQUE: VALUE LISTS

B+Tree Leaf Node



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



INTRA-NODE SEARCH

Approach #1: Linear

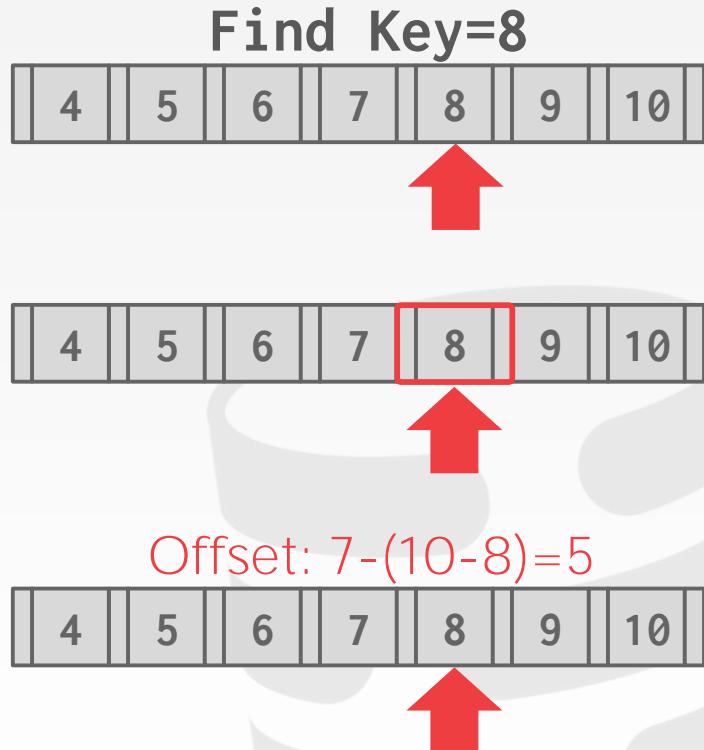
→ Scan node keys from beginning to end.

Approach #2: Binary

→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.



OPTIMIZATIONS

Prefix Compression

Suffix Truncation

Bulk Insert

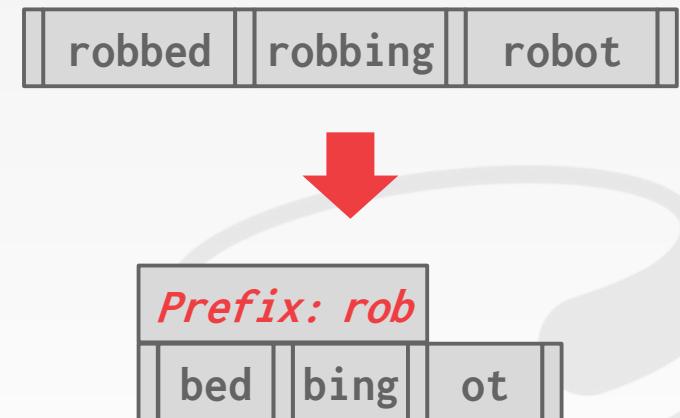
Pointer Swizzling



PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

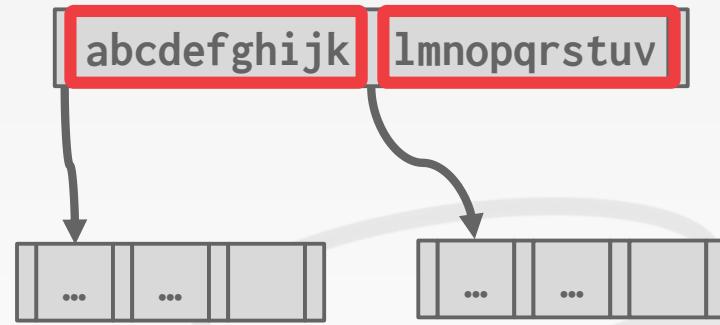
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

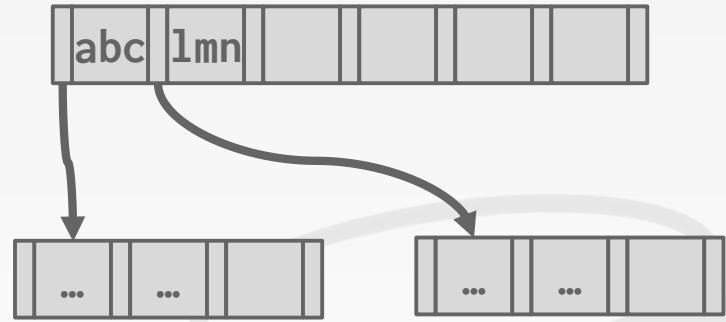
Store a minimum prefix that is needed to correctly route probes into the index.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.

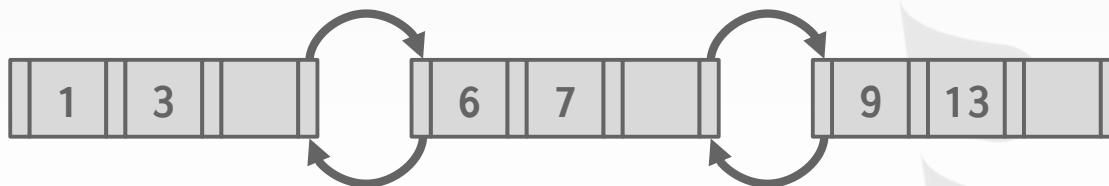


BULK INSERT

The fastest/best way to build a B+ Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13

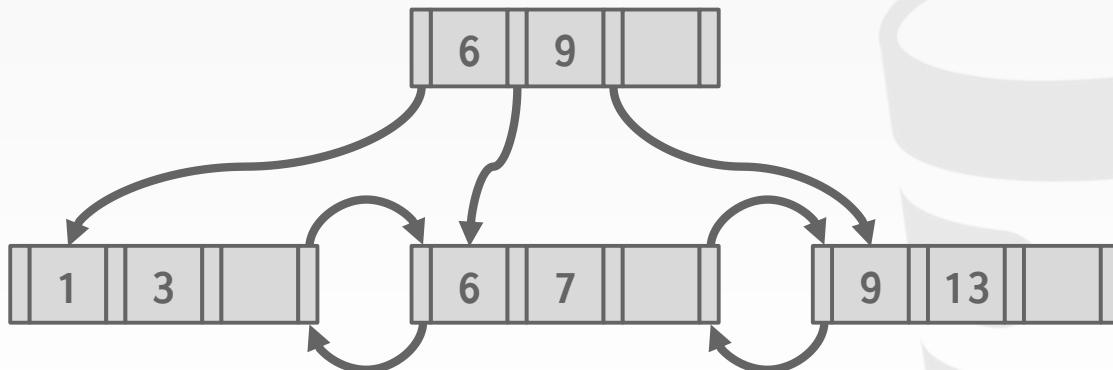


BULK INSERT

The fastest/best way to build a B+ Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

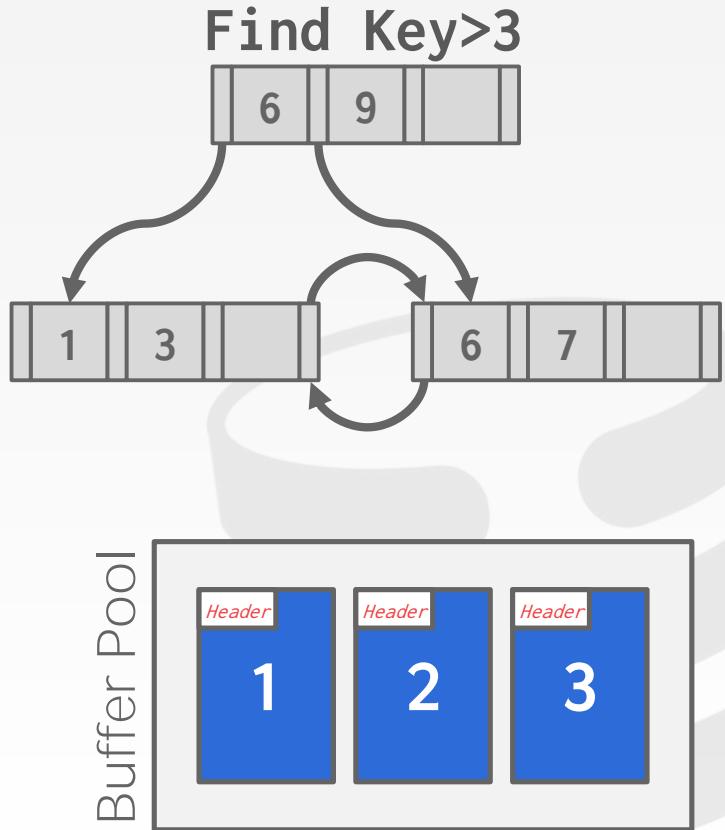
Sorted Keys: 1, 3, 6, 7, 9, 13



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

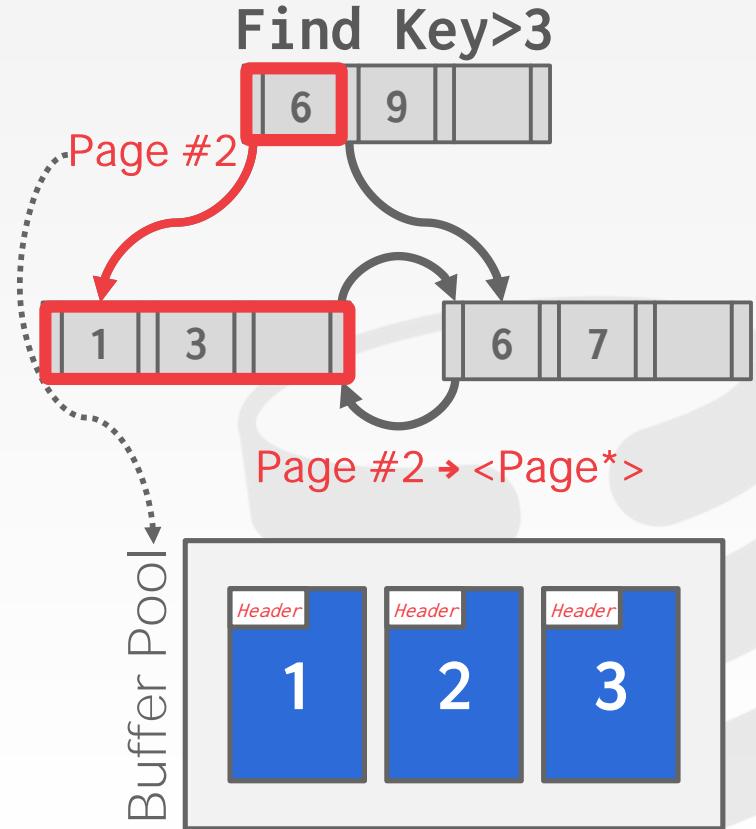
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

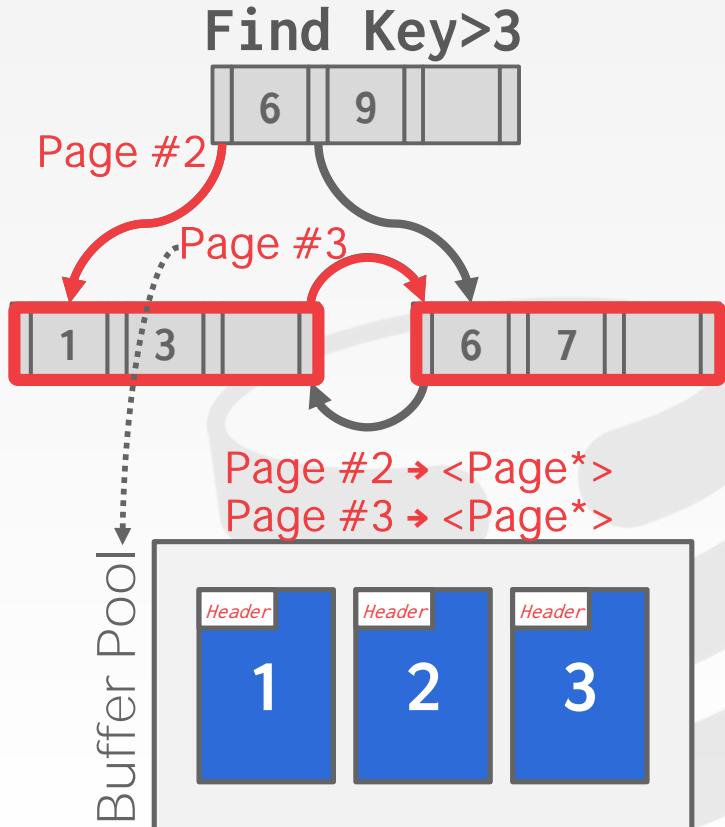
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

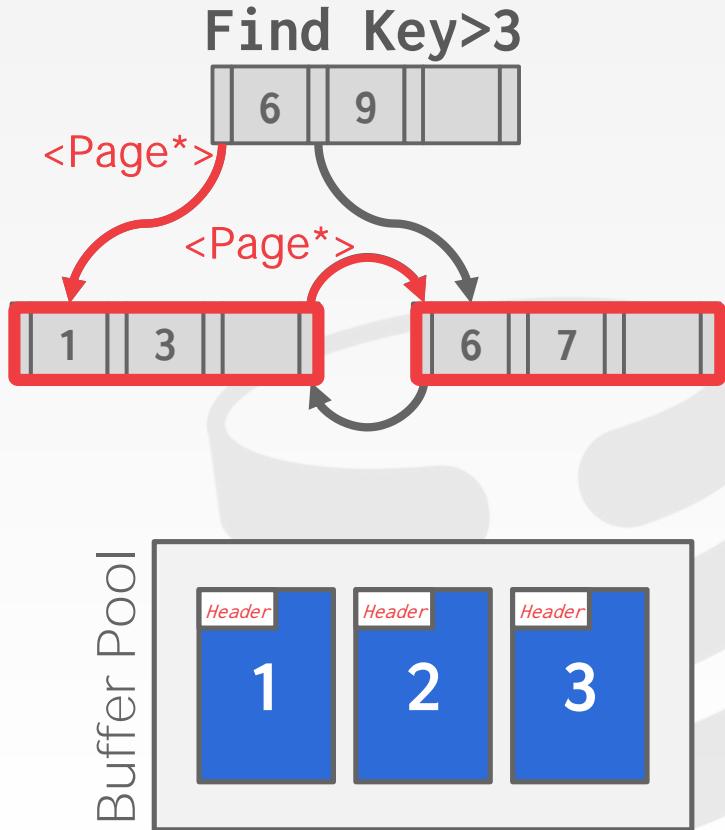
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



CONCLUSION

The venerable B+Tree is always a good choice for your DBMS.



NEXT CLASS

More B+ Trees

Tries / Radix Trees

Inverted Indexes



08 |

Tree Indexes — Part II



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

UPCOMING DATABASE EVENTS

Vertica Talk

- Monday Sep 23rd @ 4:30pm
- GHC 8102



TODAY'S AGENDA

More B+Trees

Additional Index Magic

Tries / Radix Trees

Inverted Indexes



B + TREE: DUPLICATE KEYS

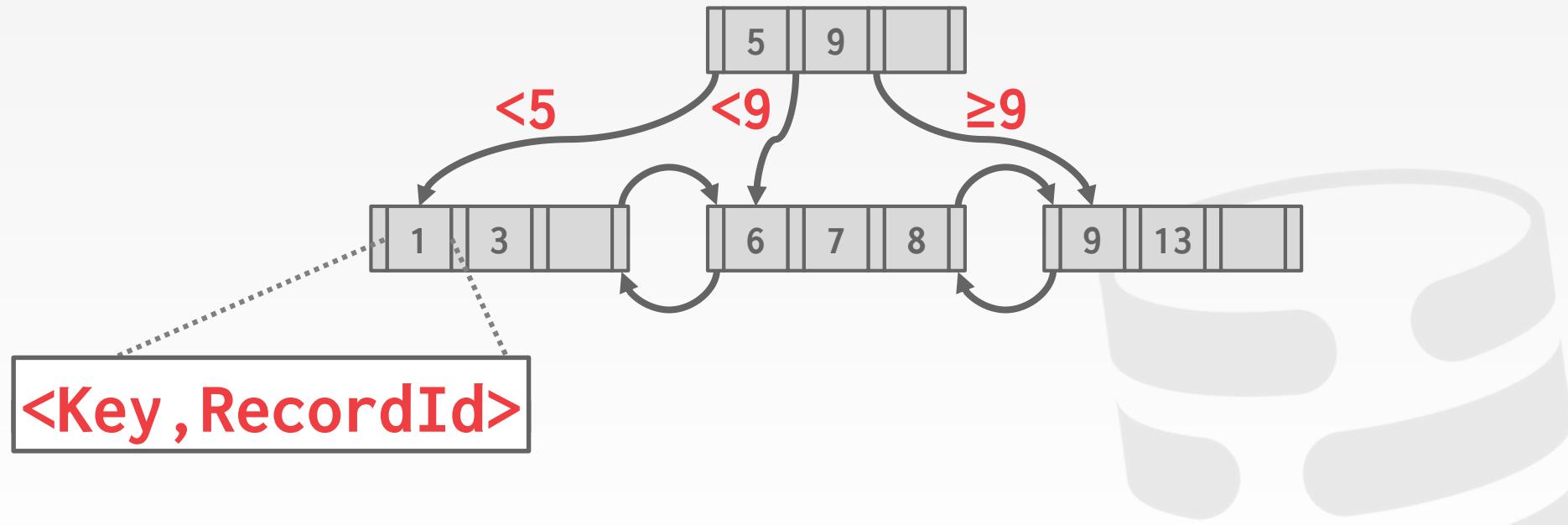
Approach #1: Append Record Id

- Add the tuple's unique record id as part of the key to ensure that all keys are unique.
- The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes

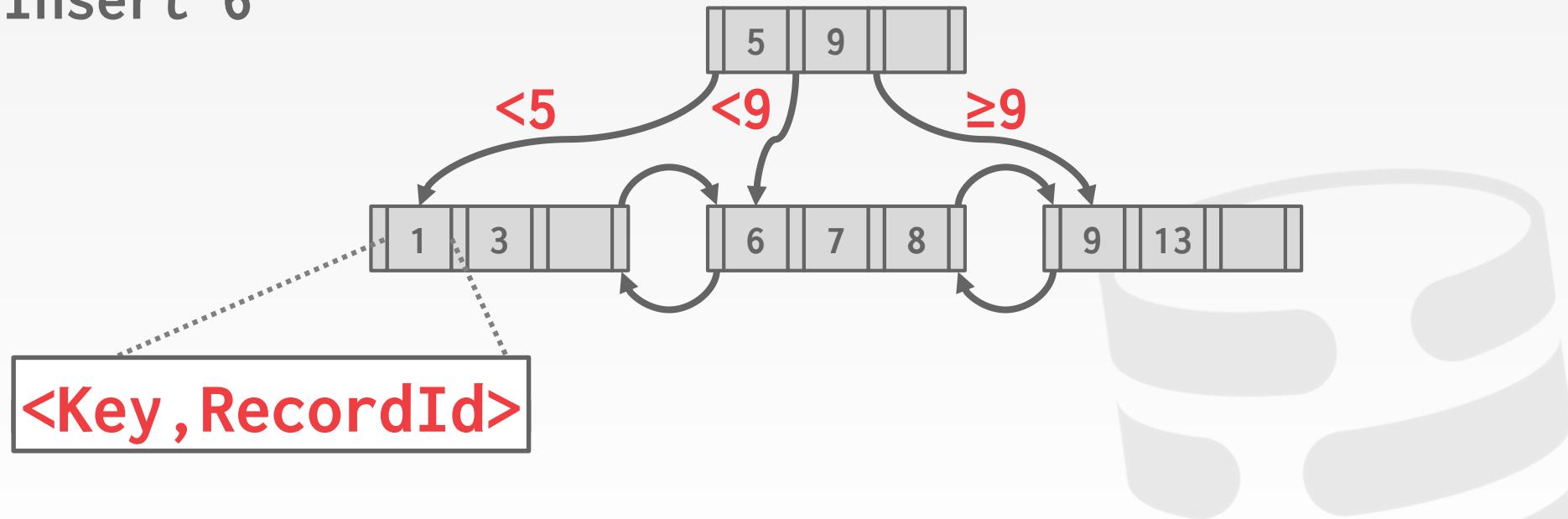
- Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.
- This is more complex to maintain and modify.

B+TREE: APPEND RECORD ID



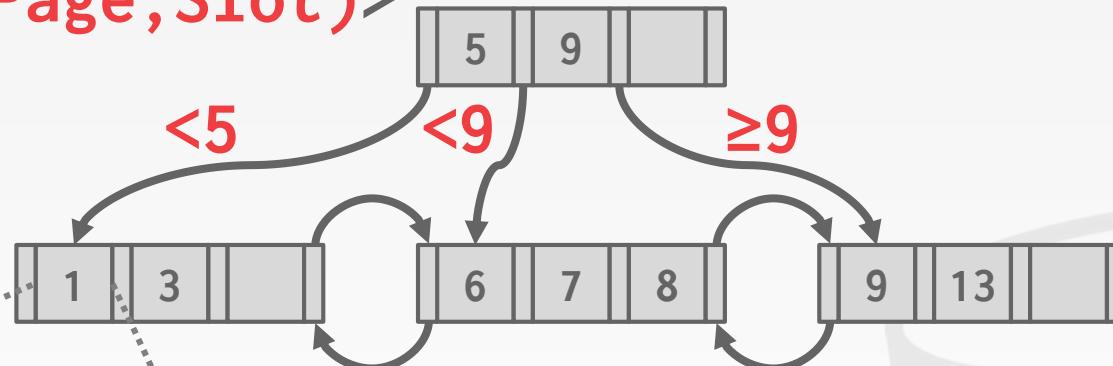
B+TREE: APPEND RECORD ID

Insert 6



B+TREE: APPEND RECORD ID

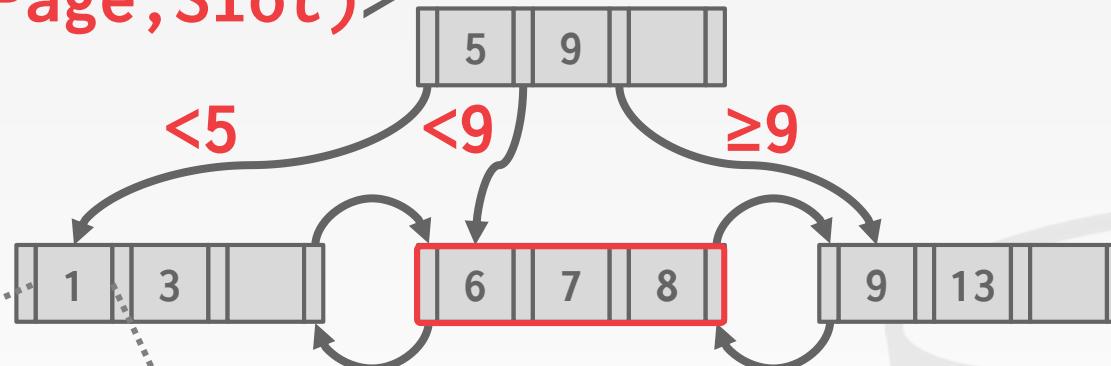
Insert <6, (Page, Slot)>



<Key , RecordId>

B+TREE: APPEND RECORD ID

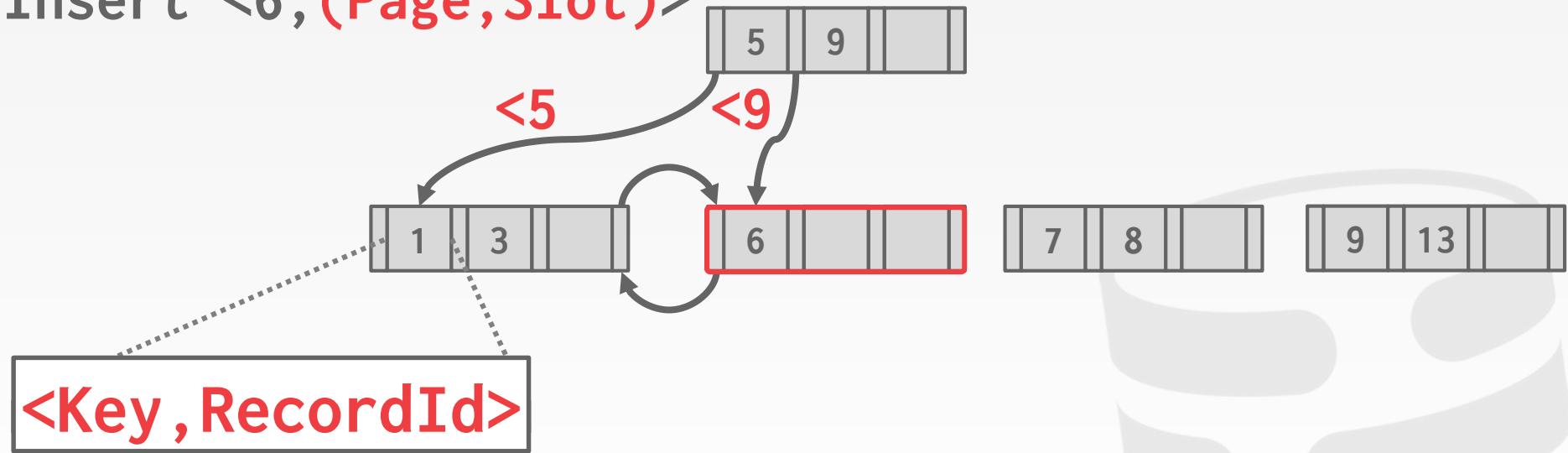
Insert <6, (Page, Slot)>



<Key , RecordId>

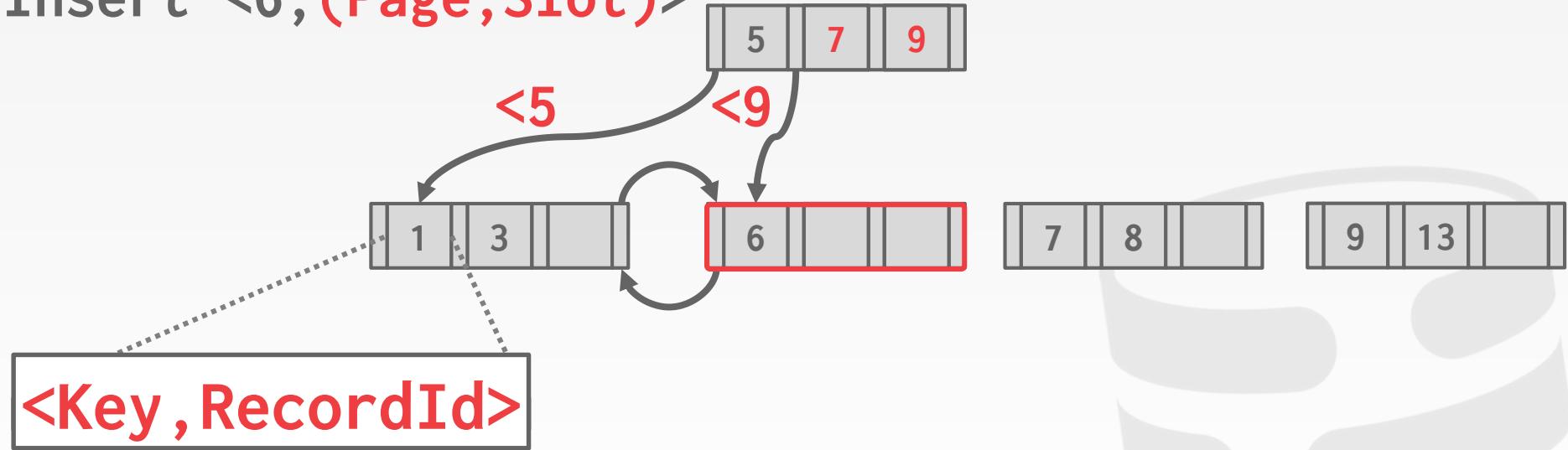
B+TREE: APPEND RECORD ID

Insert <6, (Page, Slot)>



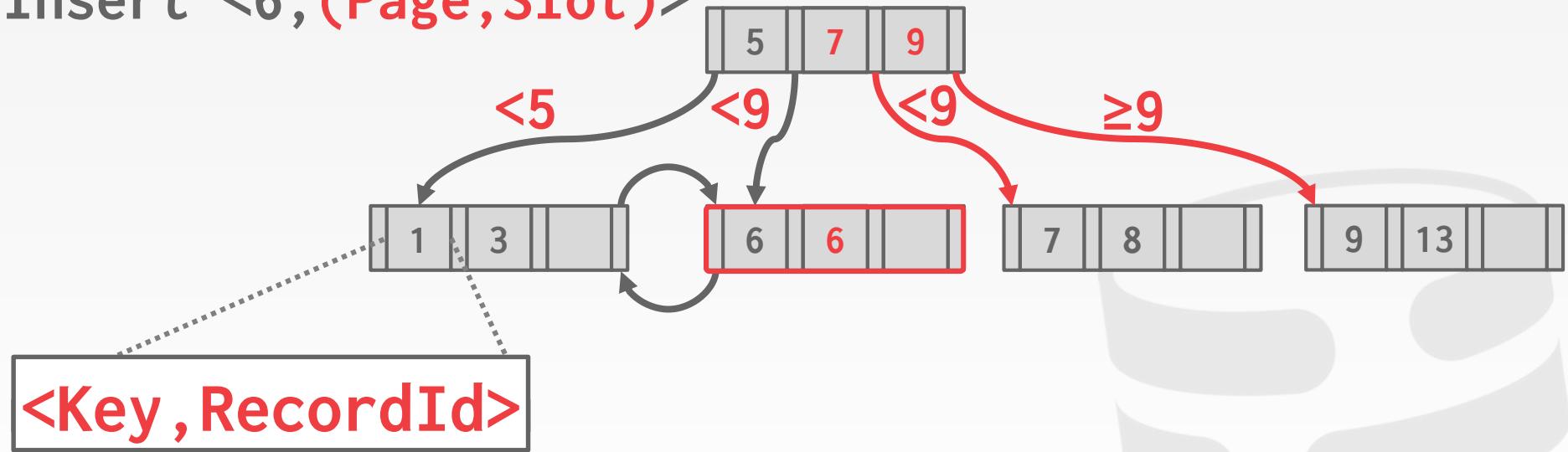
B+TREE: APPEND RECORD ID

Insert <6, (Page, Slot)>



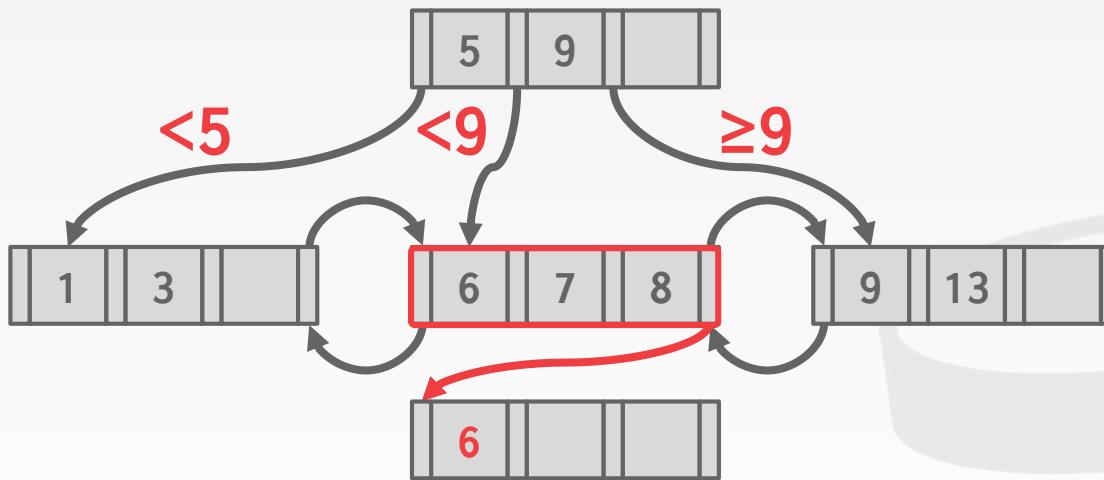
B+TREE: APPEND RECORD ID

Insert <6, (Page, Slot)>



B+TREE: OVERFLOW LEAF NODES

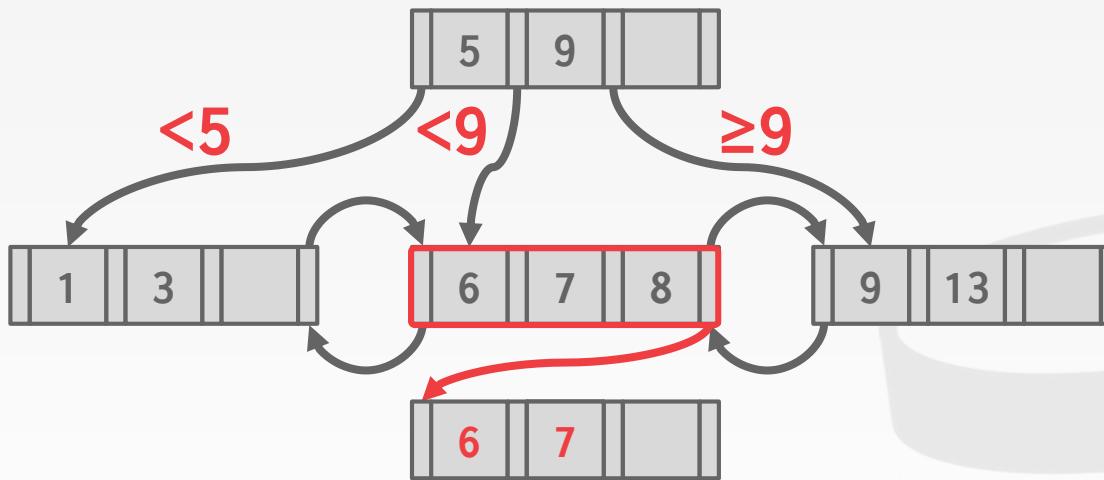
Insert 6



B+TREE: OVERFLOW LEAF NODES

Insert 6

Insert 7

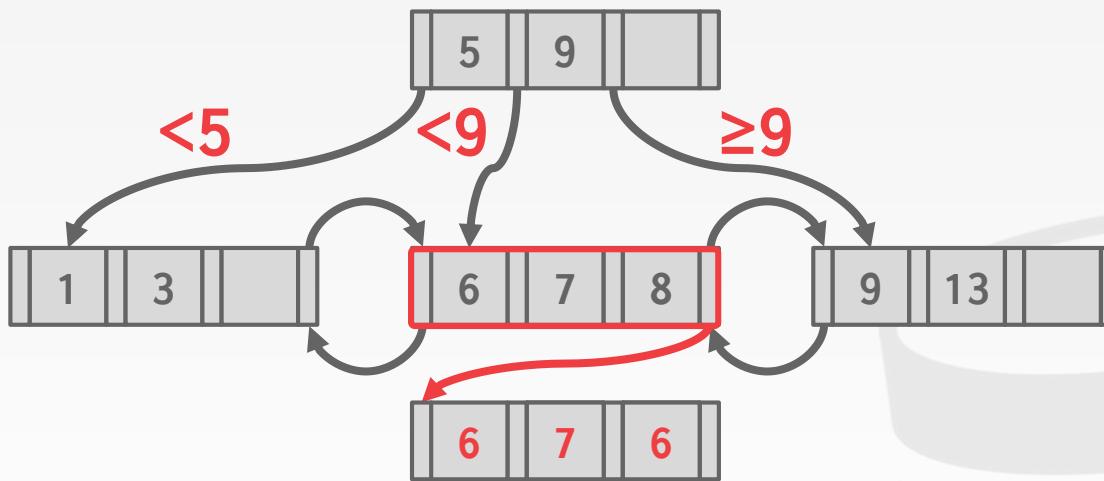


B+TREE: OVERFLOW LEAF NODES

Insert 6

Insert 7

Insert 6



DEMO

B+ Tree vs. Hash Indexes

Table Clustering



IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

- Primary Keys
- Unique Constraints

```
CREATE TABLE foo (
    id SERIAL PRIMARY KEY,
    val1 INT NOT NULL,
    val2 VARCHAR(32) UNIQUE
);
```

```
CREATE UNIQUE INDEX foo_pkey
    ON foo (id);
```

```
CREATE UNIQUE INDEX foo_val2_key
    ON foo (val2);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

- Primary Keys
- Unique Constraints

```
CREATE TABLE foo (
    id SERIAL PRIMARY KEY,
    val1 INT NOT NULL,
    val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
    id INT REFERENCES foo (val1),
    val VARCHAR(32)
);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

- Primary Keys
- Unique Constraints

```
CREATE INDEX foo_val1_key  
ON foo (val1);
```

```
CREATE TABLE foo (  
    id SERIAL PRIMARY KEY,  
    val1 INT NOT NULL,  
    val2 VARCHAR(32) UNIQUE  
);
```

```
CREATE TABLE bar (  
    id INT REFERENCES foo (val1),  
    val VARCHAR(32)  
);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

- Primary Keys
- Unique Constraints

```
CREATE INDEX foo_val1_key  
ON foo (val1);
```

```
CREATE TABLE foo (  
    id SERIAL PRIMARY KEY,  
    val1 INT NOT NULL,  
    val2 VARCHAR(32) UNIQUE  
);
```

```
CREATE TABLE bar (  
    id INT REFERENCES foo (val1),  
    val VARCHAR(32)  
);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

- Primary Keys
- Unique Constraints

```
CREATE TABLE foo (
    id SERIAL PRIMARY KEY,
    val1 INT NOT NULL UNIQUE,
    val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
    id INT REFERENCES foo (val1),
    val VARCHAR(32)
);
```

PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
WHERE c = 'WuTang';
```

One common use case is to partition indexes by date ranges.
→ Create a separate index per month, year.



PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.
→ Create a separate index per month, year.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
WHERE c = 'WuTang';
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```

COVERING INDEXES

If all the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

```
CREATE INDEX idx_foo  
ON foo (a, b);
```

```
SELECT b FROM foo  
WHERE a = 123;
```

This reduces contention on the DBMS's buffer pool resources.

COVERING INDEXES

If all the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

This reduces contention on the DBMS's buffer pool resources.

```
CREATE INDEX idx_foo  
ON foo (a, b);
```

```
SELECT b FROM foo  
WHERE a = 123;
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
INCLUDE (c)
```

These extra columns are only stored in the leaf nodes and are not part of the search key.



INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
    ON foo(a, b)  
    INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
    AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
    ON foo(a, b)  
    INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
    AND c = 'WuTang';
```

FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users  
WHERE EXTRACT(dow  
    FROM login) = 2;
```

```
CREATE INDEX idx_user_login  
ON users (login);
```

FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users  
WHERE EXTRACT(dow  
    FROM login) = 2;
```

```
CREATE INDEX X_user_login  
ON users(X_login);
```

FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users  
WHERE EXTRACT(dow  
  ↪ FROM login) = 2;
```

```
CREATE INDEX X_idx_user_login  
ON users X_login);
```

```
CREATE INDEX idx_user_login  
ON users (EXTRACT(dow FROM login));
```

FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users  
WHERE EXTRACT(dow  
    ↪ FROM login) = 2;
```

```
CREATE INDEX X_idx_user_login  
ON users X_login;
```

```
CREATE INDEX idx_user_login  
ON users (EXTRACT(dow FROM login));
```

FUNCTIONAL/EXPRESSION INDEXES

An index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users  
WHERE EXTRACT(dow  
    ↪ FROM login) = 2;
```

```
CREATE INDEX X_idx_user_login  
ON users X_login);
```

```
CREATE INDEX idx_user_login  
ON users (EXTRACT(dow FROM login));
```

```
CREATE INDEX idx_user_login  
ON foo (login)  
WHERE EXTRACT(dow FROM login) = 2;
```

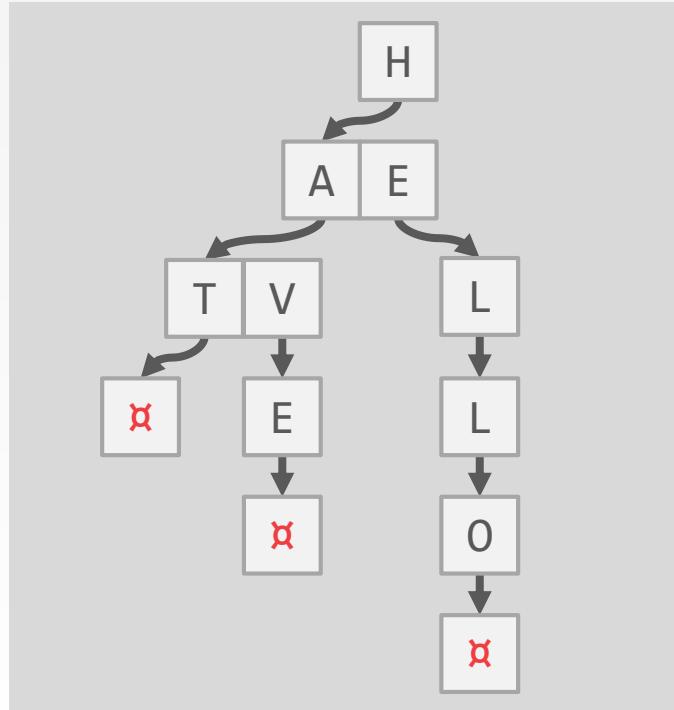
OBSERVATION

The inner node keys in a B+Tree cannot tell you whether a key exists in the index. You must always traverse to the leaf node.

This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.

TRIE INDEX

Keys: HELLO, HAT, HAVE

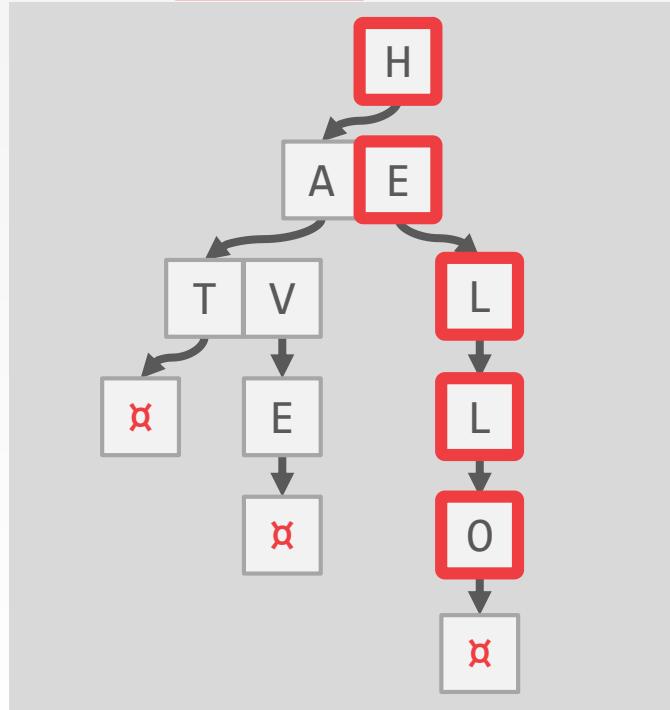


Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as *Digital Search Tree*, *Prefix Tree*.

TRIE INDEX

Keys: **HELLO**, HAT, HAVE



Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as *Digital Search Tree*, *Prefix Tree*.

TRIE INDEX PROPERTIES

Shape only depends on key space and lengths.

- Does not depend on existing keys or insertion order.
- Does not require rebalancing operations.

All operations have **O(k)** complexity where k is the length of the key.

- The path to a leaf node represents the key of the leaf
- Keys are stored implicitly and can be reconstructed from paths.

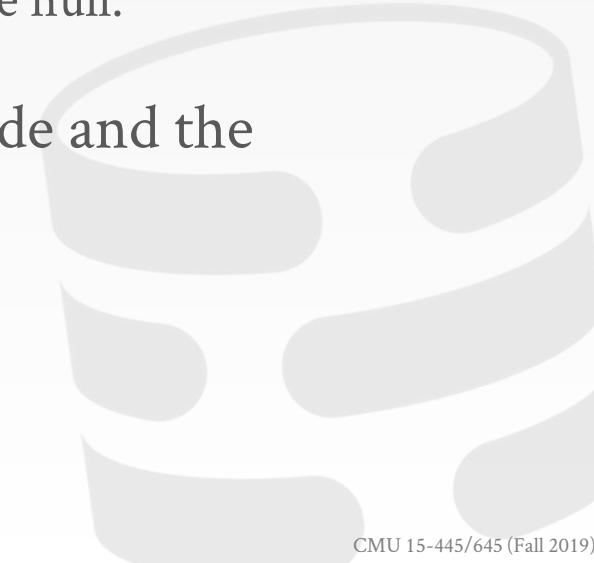
TRIE KEY SPAN

The span of a trie level is the number of bits that each partial key / digit represents.

- If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

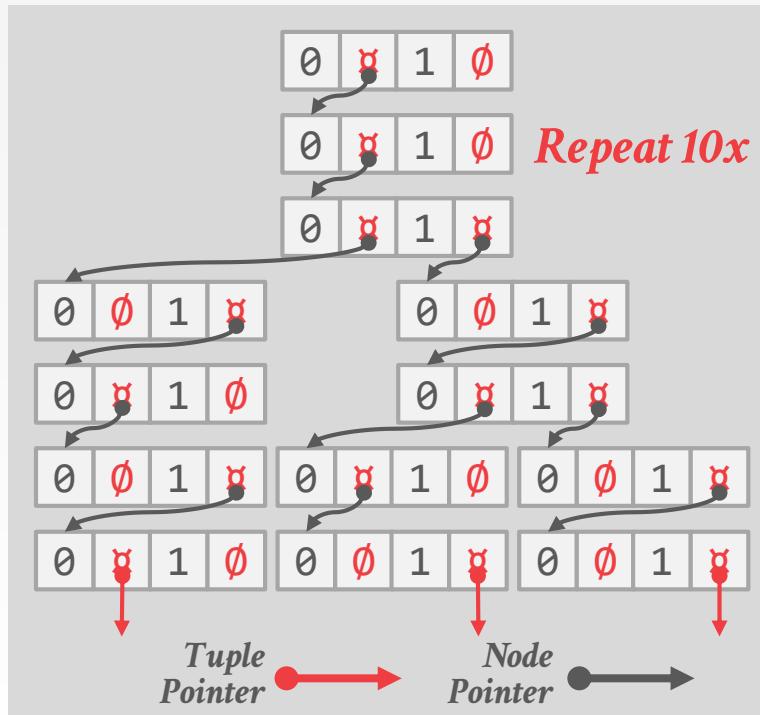
This determines the fan-out of each node and the physical height of the tree.

- n -way Trie = Fan-Out of n



TRIE KEY SPAN

1-bit Span Trie



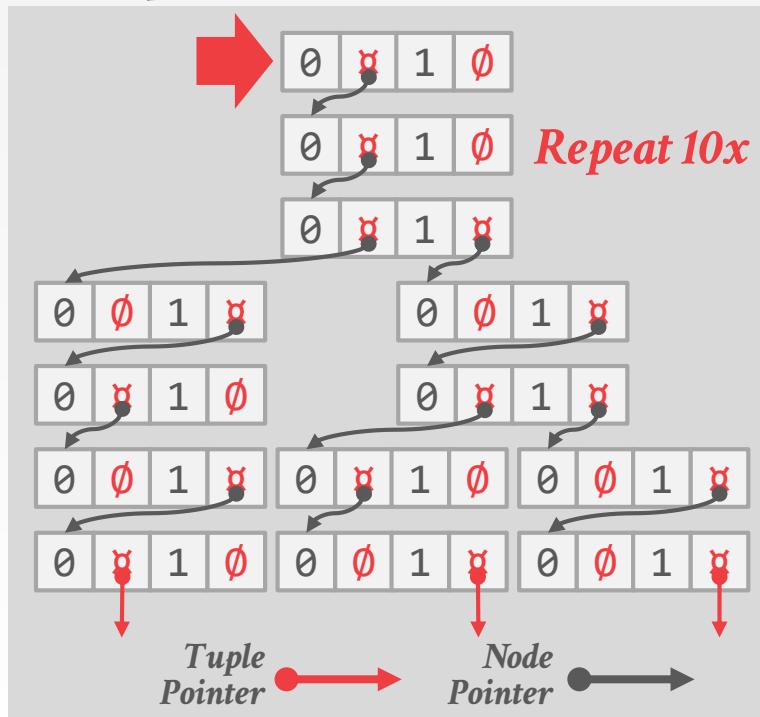
K10 → 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

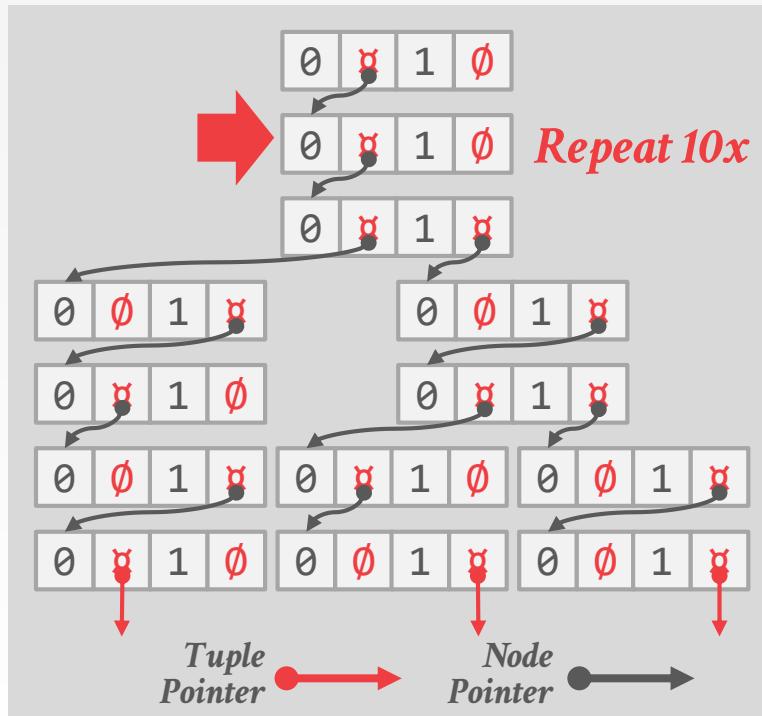


Red arrow pointing to the first column of the table:

K10 →	00000000	00001010
K25 →	00000000	00011001
K31 →	00000000	00011111

TRIE KEY SPAN

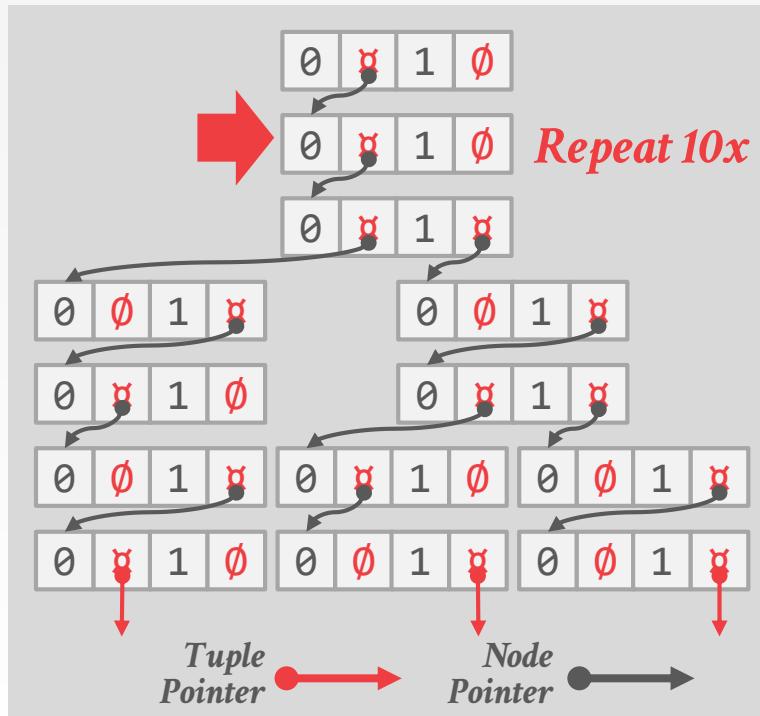
1-bit Span Trie



K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111

TRIE KEY SPAN

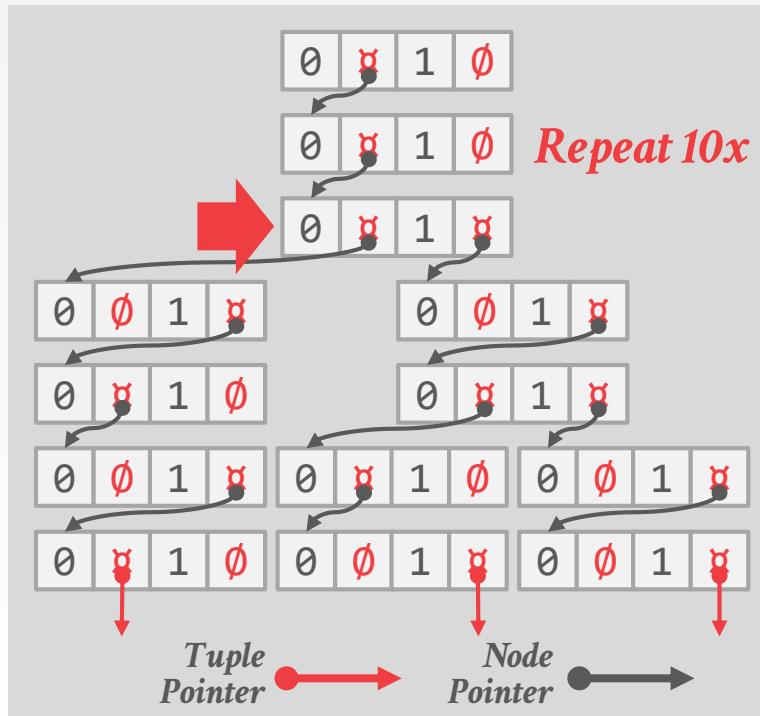
1-bit Span Trie



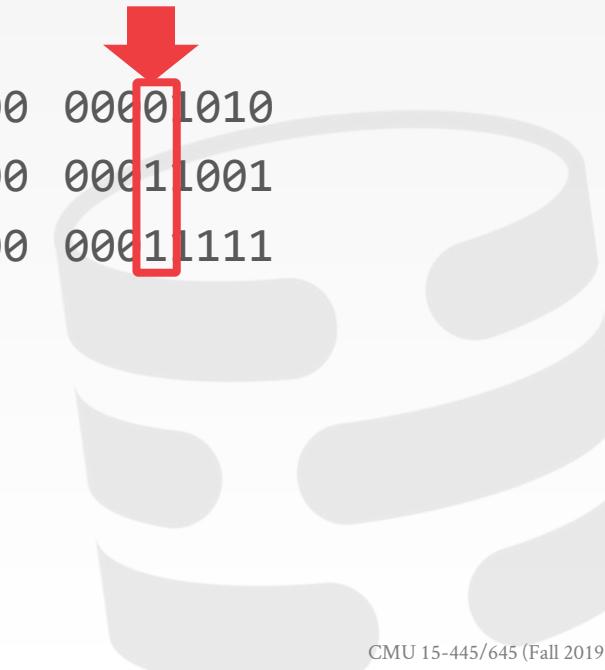
K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

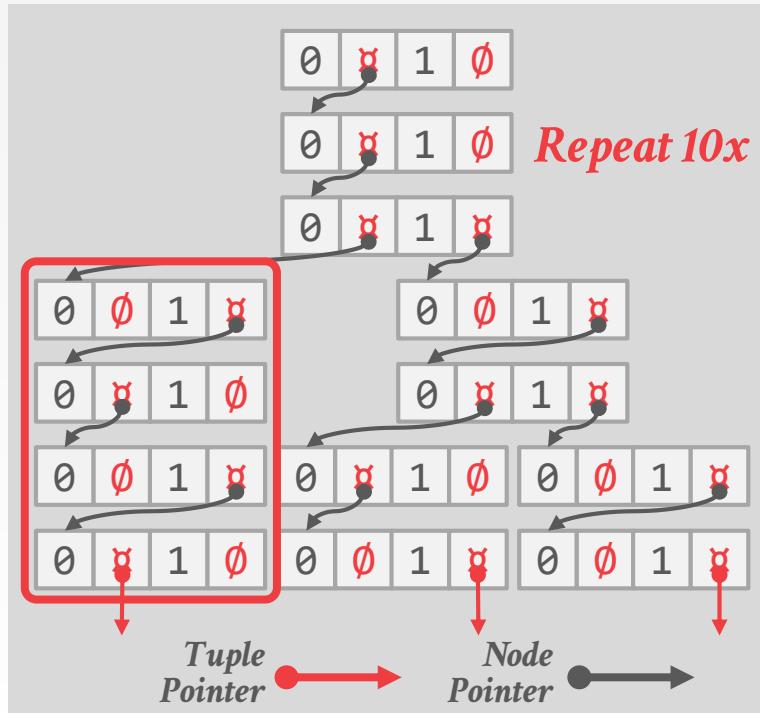


K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111



TRIE KEY SPAN

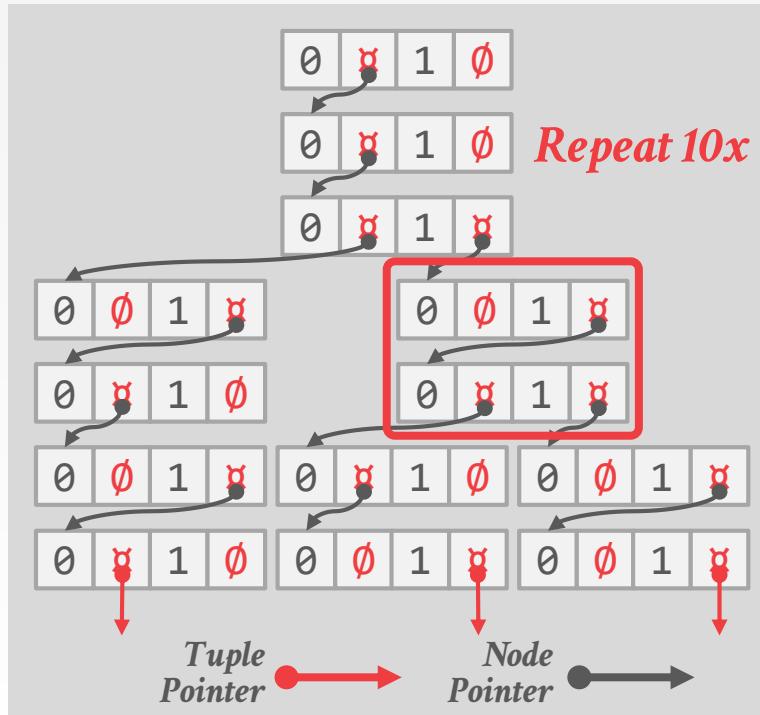
1-bit Span Trie



K10 →	00000000	00001010
K25 →	00000000	00011001
K31 →	00000000	00011111

TRIE KEY SPAN

1-bit Span Trie

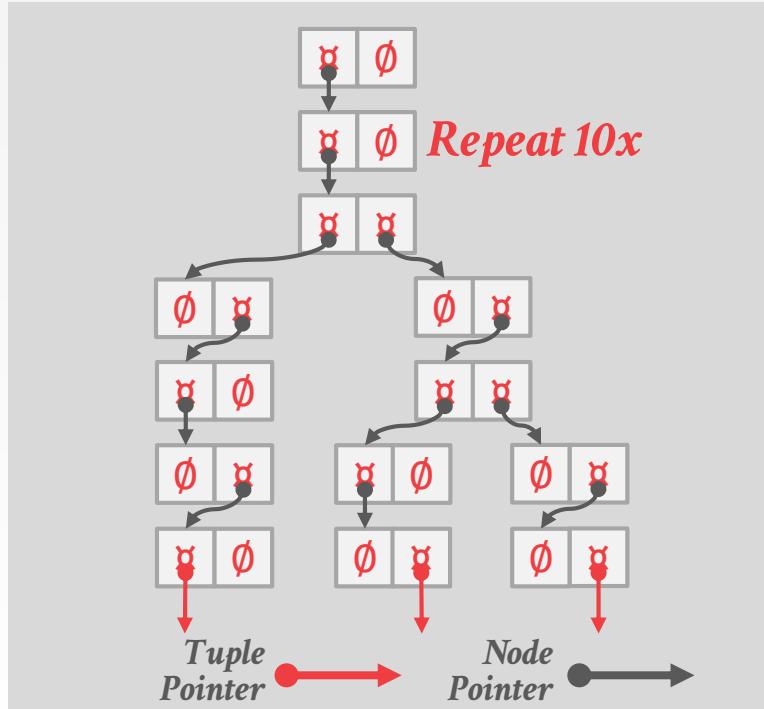


Repeat 10x

K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



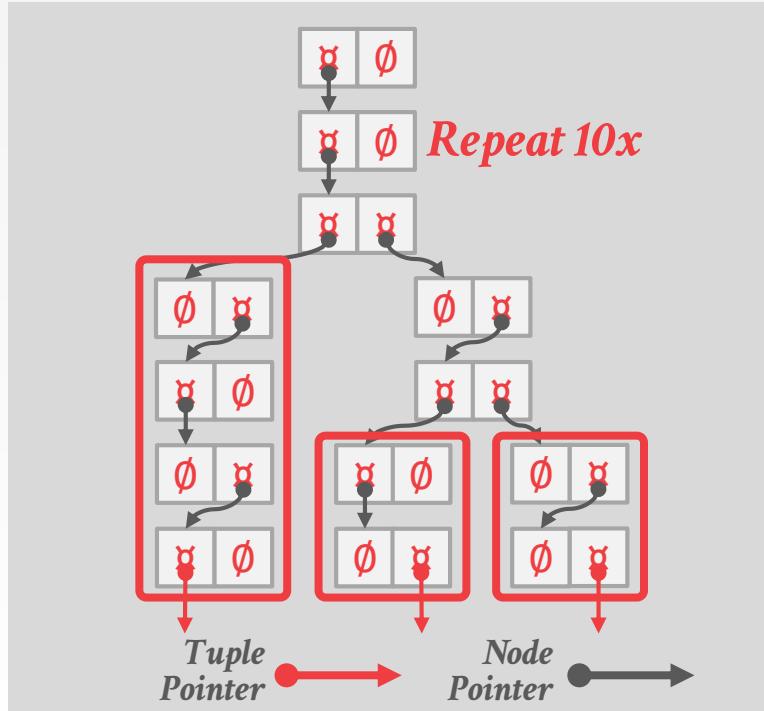
K10 → 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



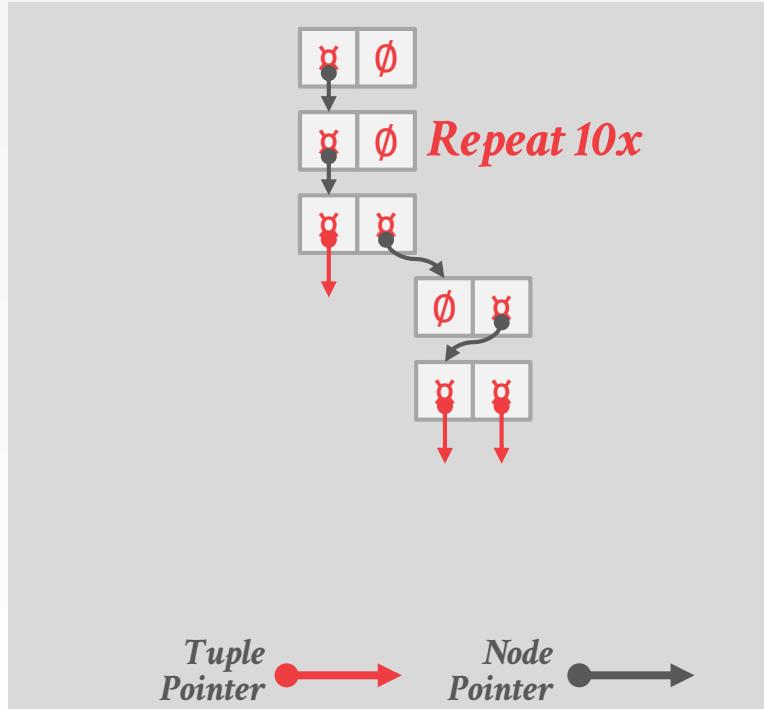
K10 → 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

RADIX TREE

1-bit Span Radix Tree

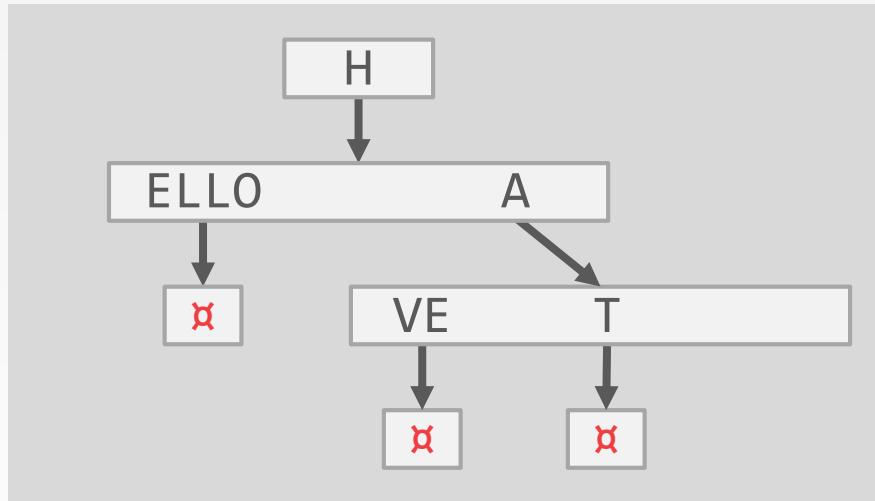


Omit all nodes with only a single child.

→ Also known as *Patricia Tree*.

Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

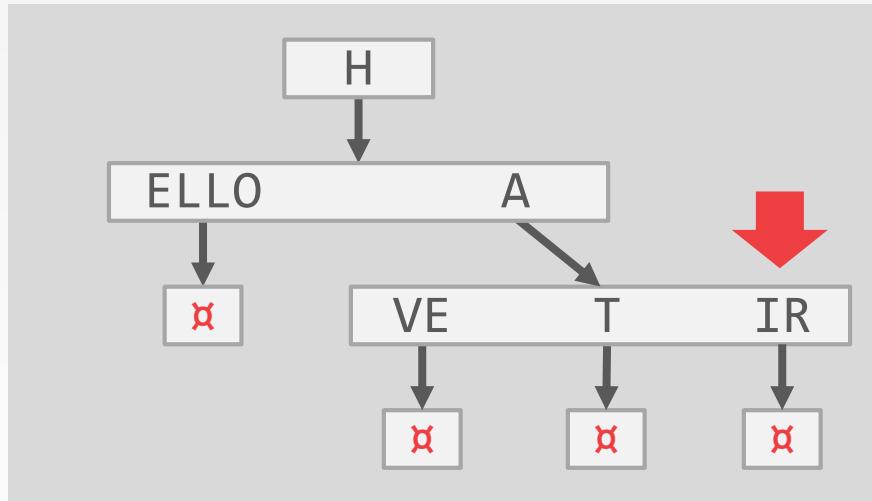
RADIX TREE: MODIFICATIONS



Insert HAIR



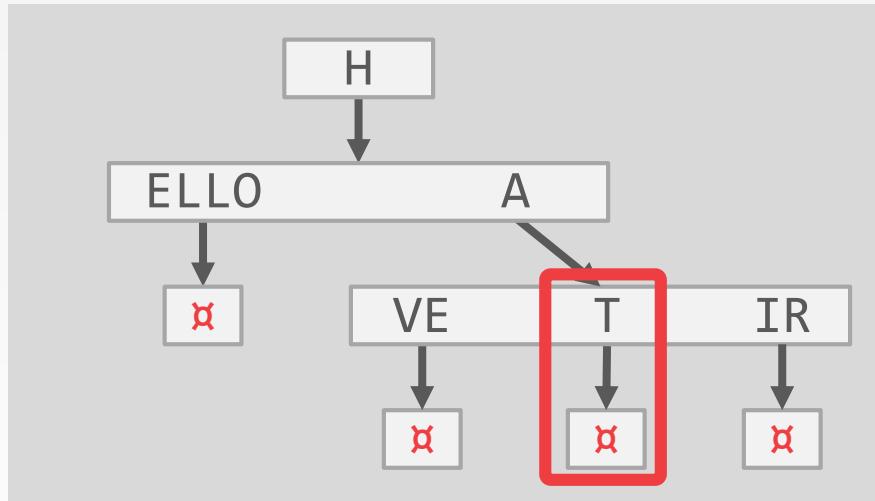
RADIX TREE: MODIFICATIONS



Insert HAIR



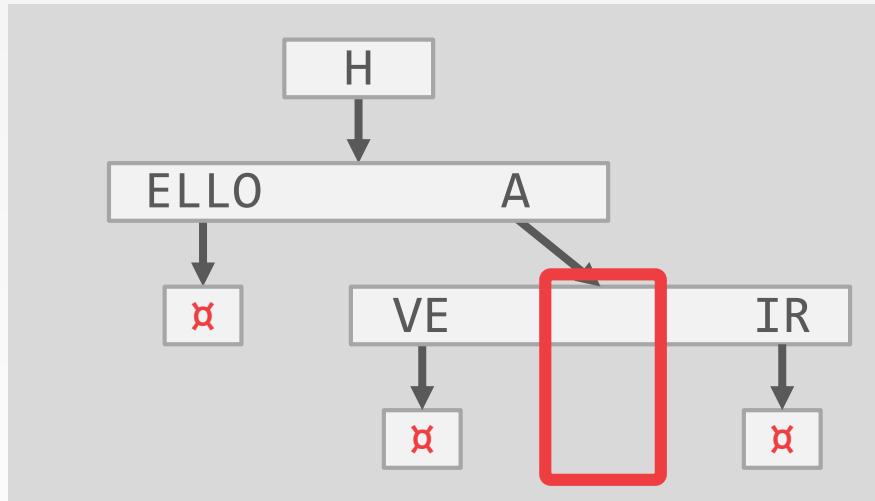
RADIX TREE: MODIFICATIONS



Insert HAIR
Delete HAT



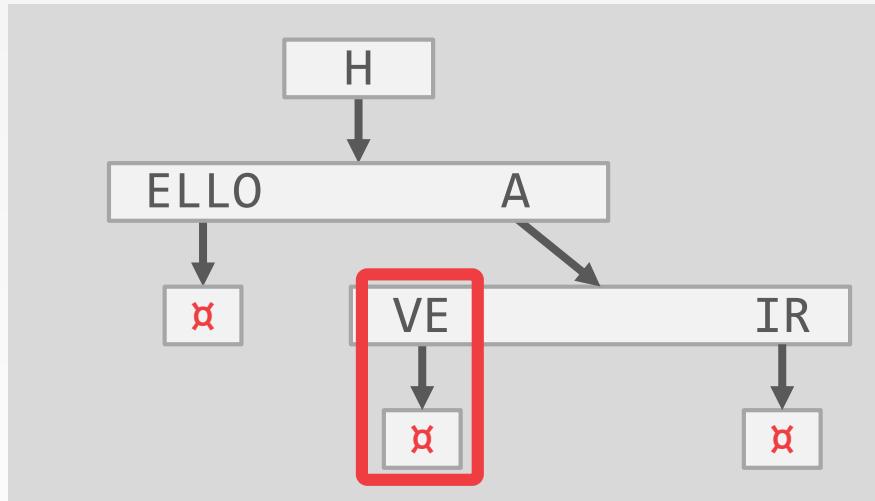
RADIX TREE: MODIFICATIONS



Insert HAIR
Delete HAT

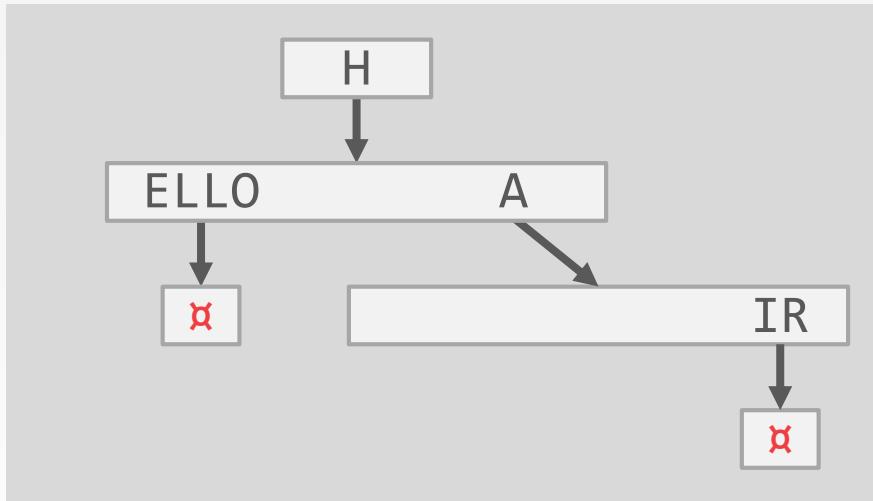


RADIX TREE: MODIFICATIONS



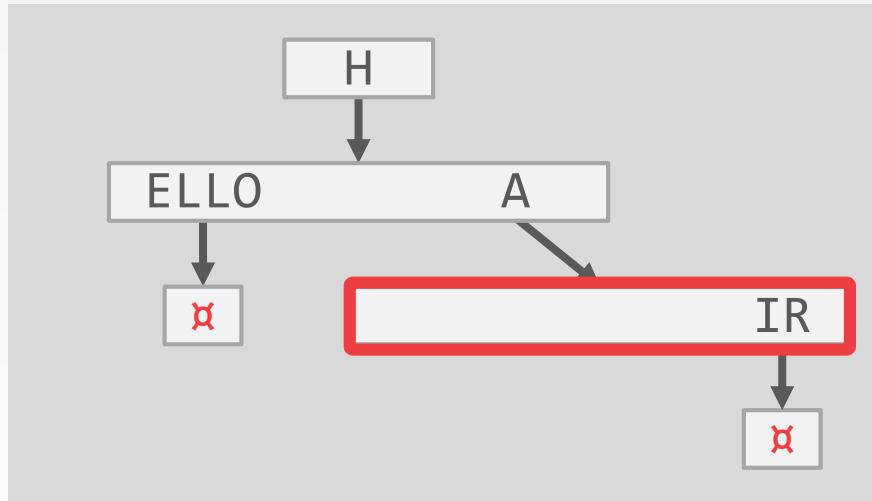
Insert HAIR
Delete HAT
Delete HAVE

RADIX TREE: MODIFICATIONS



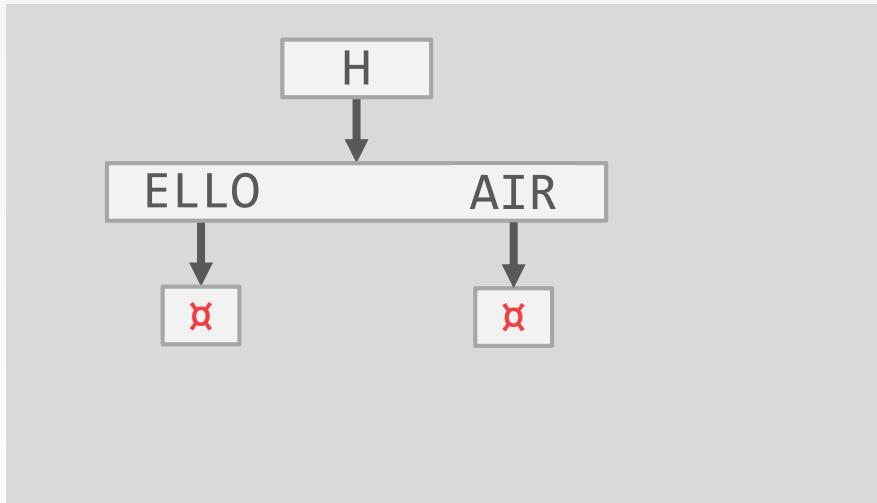
Insert HAIR
Delete HAT
Delete HAVE

RADIX TREE: MODIFICATIONS



Insert HAIR
Delete HAT
Delete HAVE

RADIX TREE: MODIFICATIONS



Insert HAIR
Delete HAT
Delete HAVE



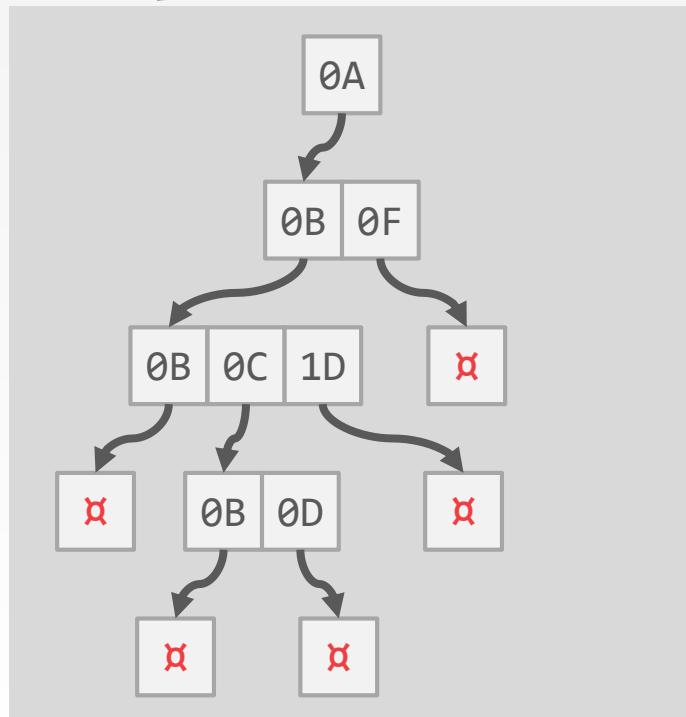
RADIX TREE: BINARY COMPARABLE KEYS

Not all attribute types can be decomposed into binary comparable digits for a radix tree.

- **Unsigned Integers:** Byte order must be flipped for little endian machines.
- **Signed Integers:** Flip two's-complement so that negative numbers are smaller than positive.
- **FLOATS:** Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.
- **Compound:** Transform each attribute separately.

RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



Int Key: 168496141



Hex Key: 0A 0B 0C 0D



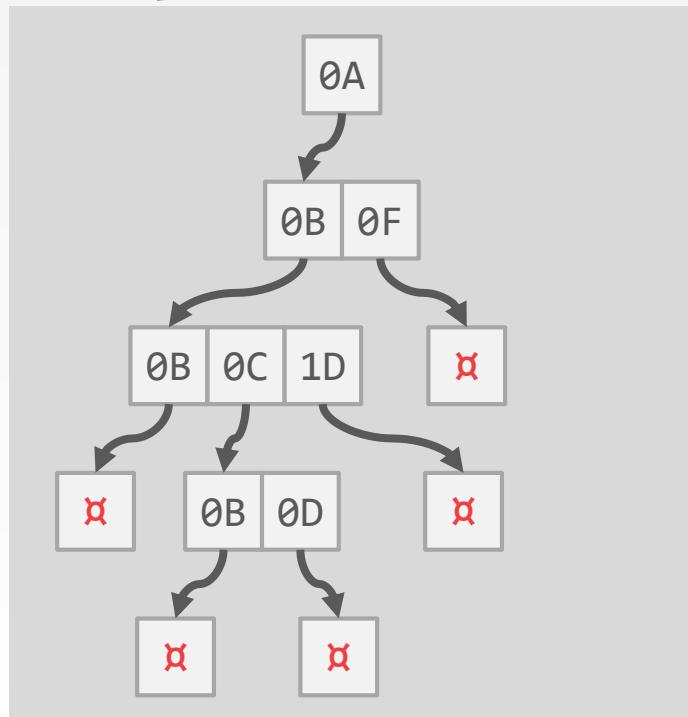
*Little
Endian*



*Big
Endian*

RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



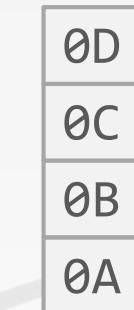
Int Key: 168496141



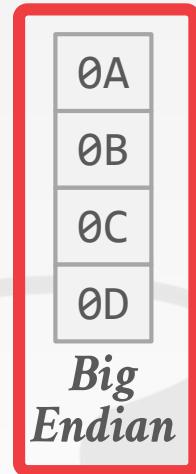
Hex Key: 0A 0B 0C 0D

Find 658205

Hex 0A 0B 1D



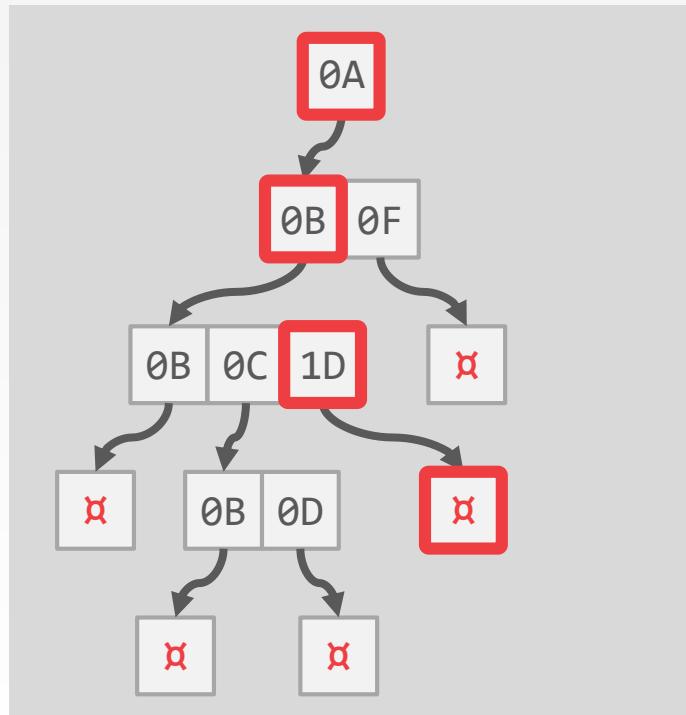
Little
Endian



Big
Endian

RADIX TREE: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



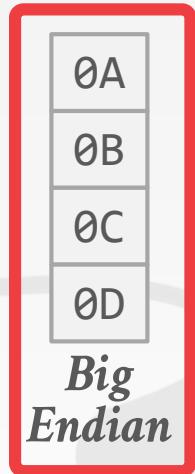
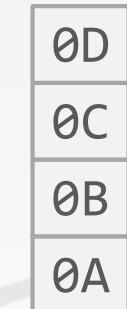
Int Key: 168496141



Hex Key: 0A 0B 0C 0D

Find 658205

Hex 0A 0B 1D



OBSERVATION

The tree indexes that we've discussed so far are useful for "point" and "range" queries:

- Find all customers in the 15217 zip code.
- Find all orders between June 2018 and September 2018.

They are not good at keyword searches:

- Find all Wikipedia articles that contain the word "Pavlo"

WIKIPEDIA EXAMPLE

```
CREATE TABLE useracct (
    userID INT PRIMARY KEY,
    userName VARCHAR UNIQUE,
    :
);
```

```
CREATE TABLE pages (
    pageID INT PRIMARY KEY,
    title VARCHAR UNIQUE,
    latest INT
    ↗ REFERENCES revisions (revID),
);
```

```
CREATE TABLE revisions (
    revID INT PRIMARY KEY,
    userID INT REFERENCES useracct (userID),
    pageID INT REFERENCES pages (pageID),
    content TEXT,
    updated DATETIME
);
```

WIKIPEDIA EXAMPLE

If we create an index on the content attribute, what does that do?

```
CREATE INDEX idx_rev_cntnt  
ON revisions (content);
```

This doesn't help our query.
Our SQL is also not correct...

```
SELECT pageID FROM revisions  
WHERE content LIKE '%Pavlo%';
```

INVERTED INDEX

An *inverted index* stores a mapping of words to records that contain those words in the target attribute.

- Sometimes called a *full-text search index*.
- Also called a *concordance* in old (like really old) times.

The major DBMSs support these natively.
There are also specialized DBMSs.



QUERY TYPES

Phrase Searches

- Find records that contain a list of words in the given order.

Proximity Searches

- Find records where two words occur within n words of each other.

Wildcard Searches

- Find records that contain words that match some pattern (e.g., regular expression).

DESIGN DECISIONS

Decision #1: What To Store

- The index needs to store at least the words contained in each record (separated by punctuation characters).
- Can also store frequency, position, and other meta-data.

Decision #2: When To Update

- Maintain auxiliary data structures to "stage" updates and then update the index in batches.



CONCLUSION

B+ Trees are still the way to go for tree indexes.

Inverted indexes are covered in [CMU 11-442](#).

We did not discuss geo-spatial tree indexes:

- Examples: R-Tree, Quad-Tree, KD-Tree
- This is covered in [CMU 15-826](#).

NEXT CLASS

How to make indexes thread-safe!



09

Index Concurrency



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 is due Fri Sept 27th @ 11:59pm

Homework #2 is due Mon Sept 30th @ 11:59pm

Project #2 will be released Mon Sept 30th



OBSERVATION

We assumed that all the data structures that we have discussed so far are single-threaded.

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.



They Don't Do This!

CONCURRENCY CONTROL

A **concurrency control** protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:

- **Logical Correctness:** Can I see the data that I am supposed to see?
- **Physical Correctness:** Is the internal representation of the object sound?

TODAY'S AGENDA

Latches Overview

Hash Table Latching

B+Tree Latching

Leaf Node Scans

Delayed Parent Updates



LOCKS VS. LATCHES

Locks

- Protects the database's logical contents from other txns.
- Held for txn duration.
- Need to be able to rollback changes.

Latches

- Protects the critical sections of the DBMS's internal data structure from other threads.
- Held for operation duration.
- Do not need to be able to rollback changes.



LOCKS VS. LATCHES

Lecture 17



Locks

- Separate...** User transactions
- Protect...** Database Contents
- During...** Entire Transactions
- Modes...** Shared, Exclusive, Update, Intention
- Deadlock** Detection & Resolution
 - ...by...** Waits-for, Timeout, Aborts
- Kept in...** Lock Manager

Latches

- Threads
- In-Memory Data Structures
- Critical Sections
- Read, Write
- Avoidance
- Coding Discipline
- Protected Data Structure

Source: [Goetz Graefe](#)

LATCH MODES

Read Mode

- Multiple threads can read the same object at the same time.
- A thread can acquire the read latch if another thread has it in read mode.

Write Mode

- Only one thread can access the object.
- A thread cannot acquire a write latch if another thread holds the latch in any mode.

Compatibility Matrix

	Read	Write
Read	✓	✗
Write	✗	✗

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: **std::mutex**

```
std::mutex m;  
:  
m.lock();  
// Do something special...  
m.unlock();
```

LATCH IMPLEMENTATIONS

Approach #2: Test-and-Set Spin Latch (TAS)

- Very efficient (single instruction to latch/unlatch)
- Non-scalable, not cache friendly
- Example: `std::atomic<T>`

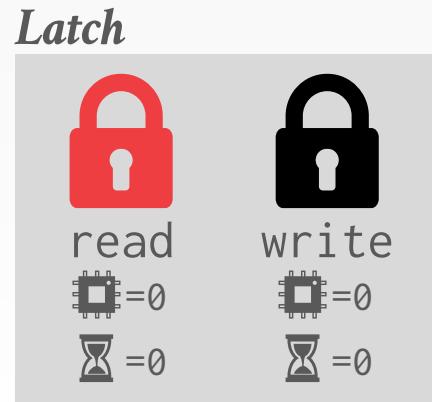
`std::atomic<bool>`

```
std::atomic_flag latch;
:
while (latch.test_and_set(...)) {
    // Retry? Yield? Abort?
}
```

LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latch

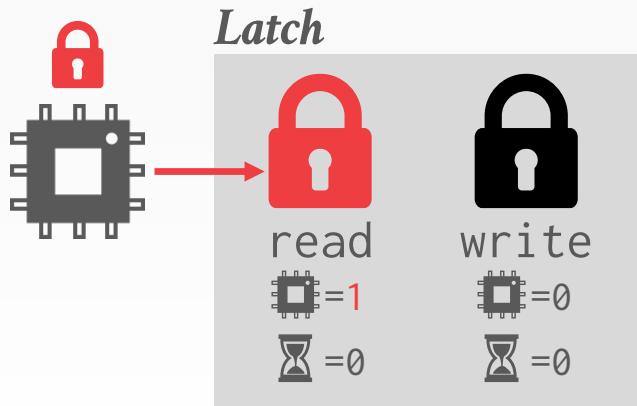
- Allows for concurrent readers
- Must manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latch

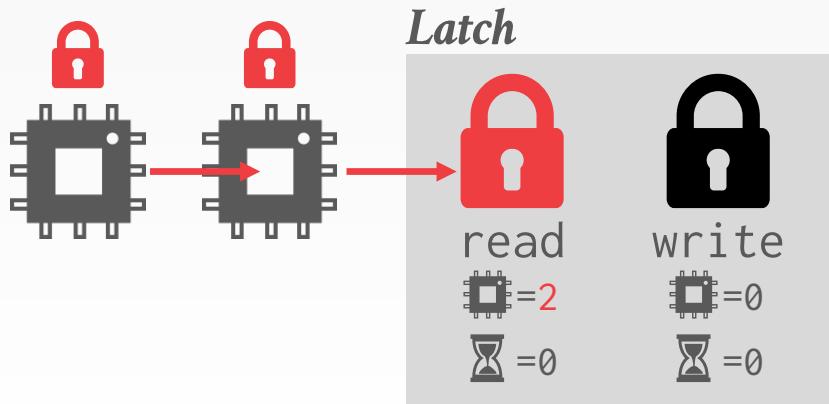
- Allows for concurrent readers
- Must manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latch

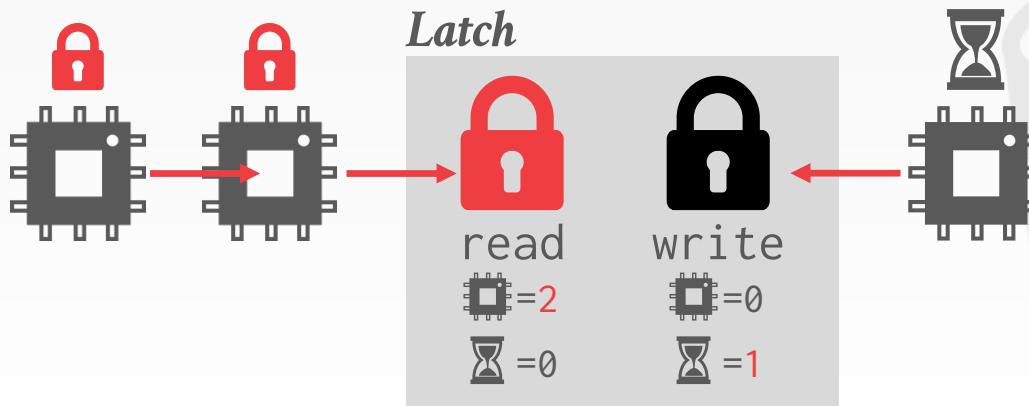
- Allows for concurrent readers
- Must manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latch

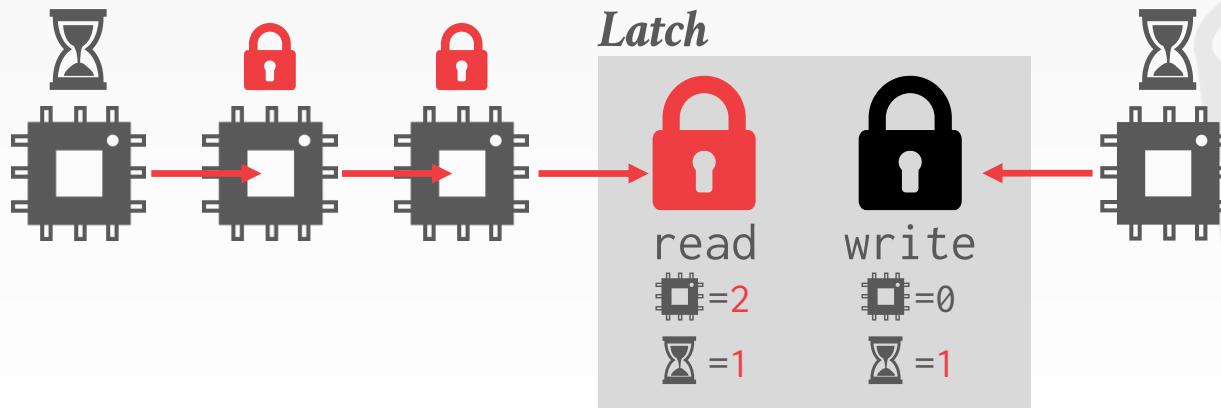
- Allows for concurrent readers
- Must manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latch

- Allows for concurrent readers
- Must manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



HASH TABLE LATCHING

Easy to support concurrent access due to the limited ways threads access the data structure.

- All threads move in the same direction and only access a single page/slot at a time.
- Deadlocks are not possible.

To resize the table, take a global latch on the entire table (i.e., in the header page).



HASH TABLE LATCHING

Approach #1: Page Latches

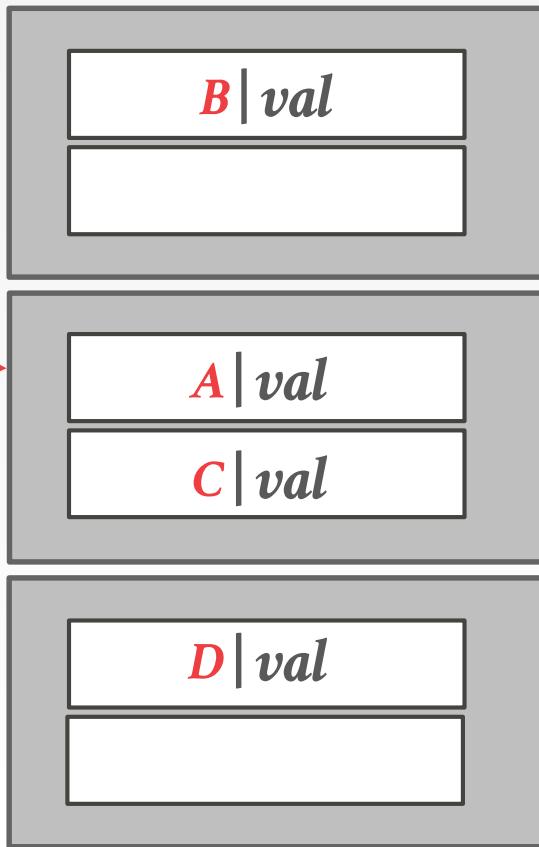
- Each page has its own reader-write latch that protects its entire contents.
- Threads acquire either a read or write latch before they access a page.

Approach #2: Slot Latches

- Each slot has its own latch.
- Can use a single mode latch to reduce meta-data and computational overhead.

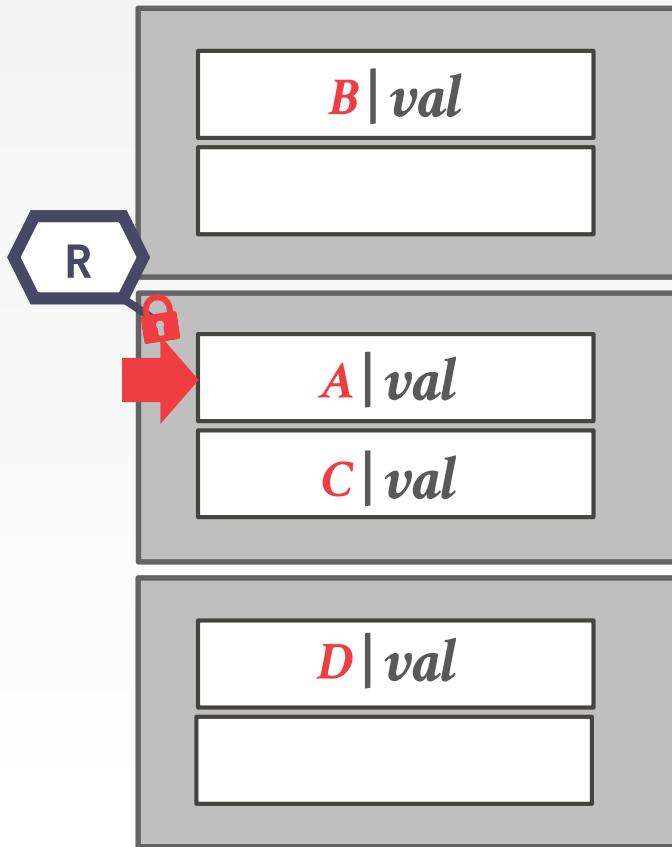
HASH TABLE – PAGE LATCHES

T₁: Find D
 $hash(D)$



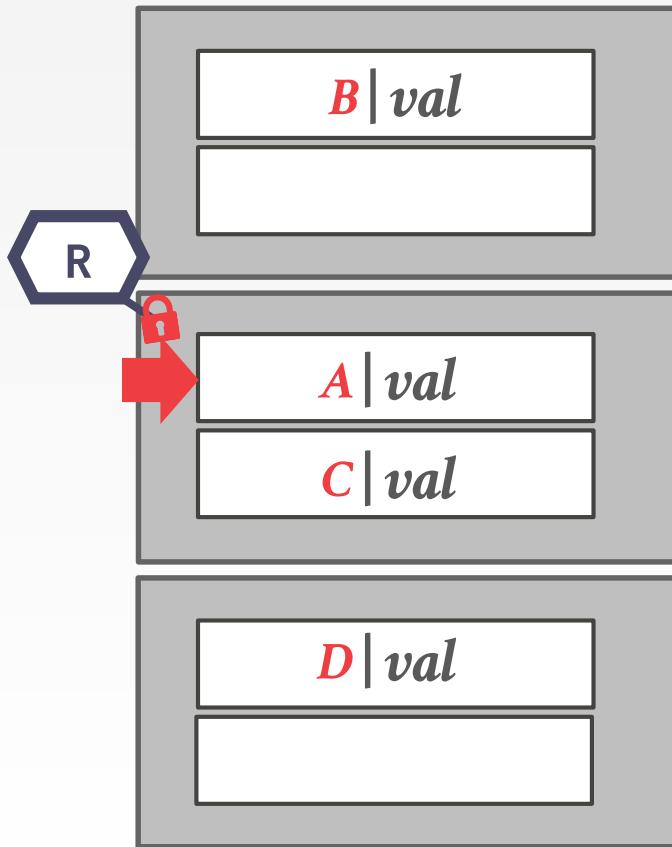
HASH TABLE – PAGE LATCHES

T₁: Find D
 $hash(D)$

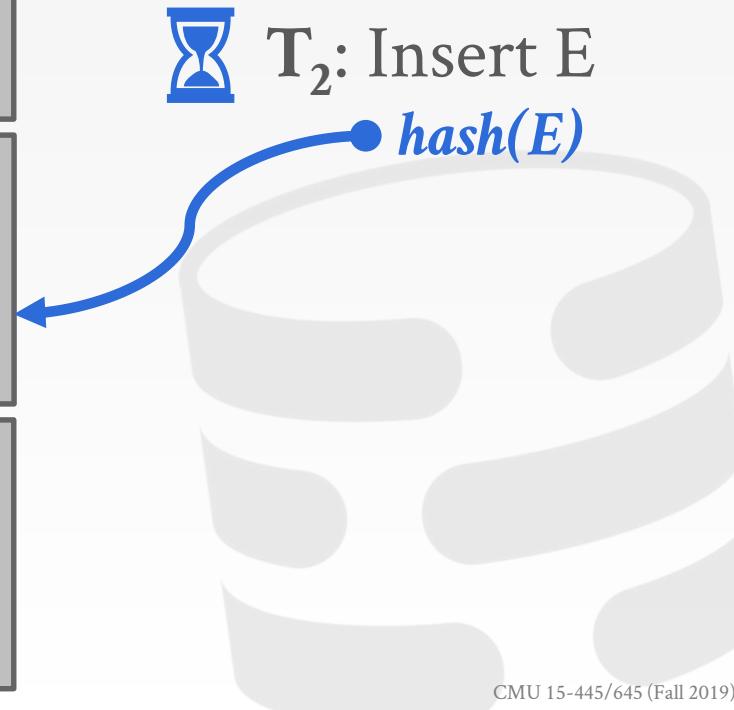


HASH TABLE – PAGE LATCHES

T₁: Find D
hash(D)

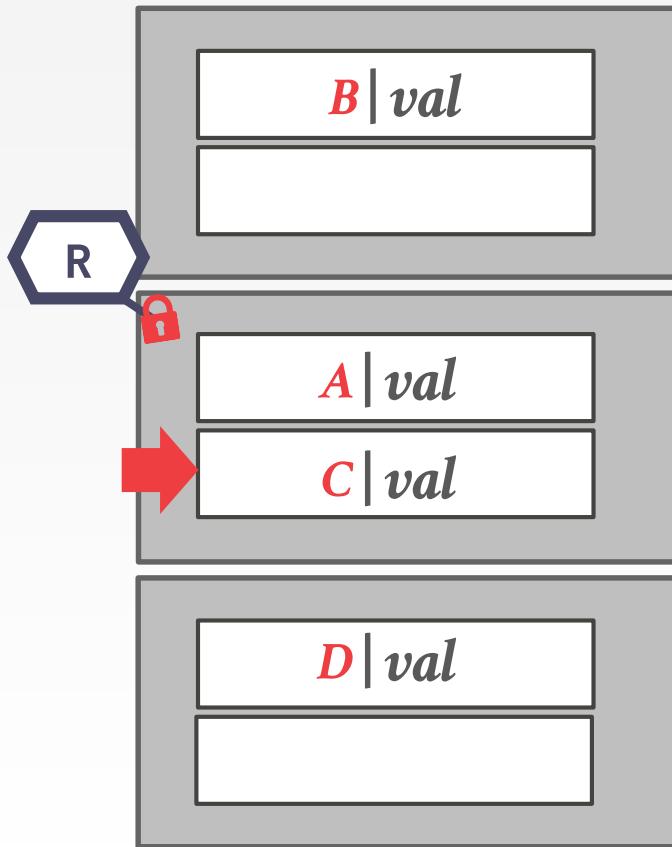


T₂: Insert E
hash(E)

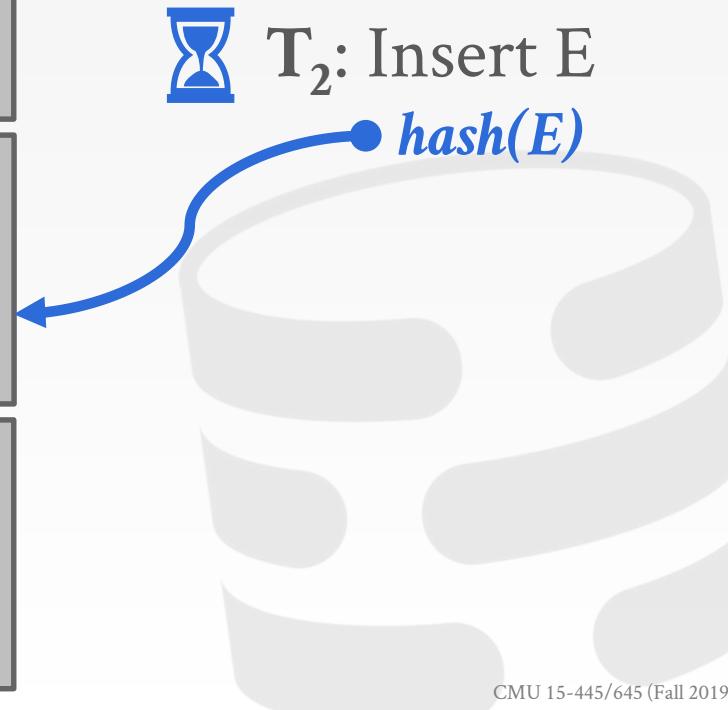


HASH TABLE – PAGE LATCHES

T₁: Find D
 $hash(D)$



T₂: Insert E
 $hash(E)$



HASH TABLE – PAGE LATCHES

It's safe to release the latch on Page #1.

T₁: Find D
 $\text{hash}(D)$

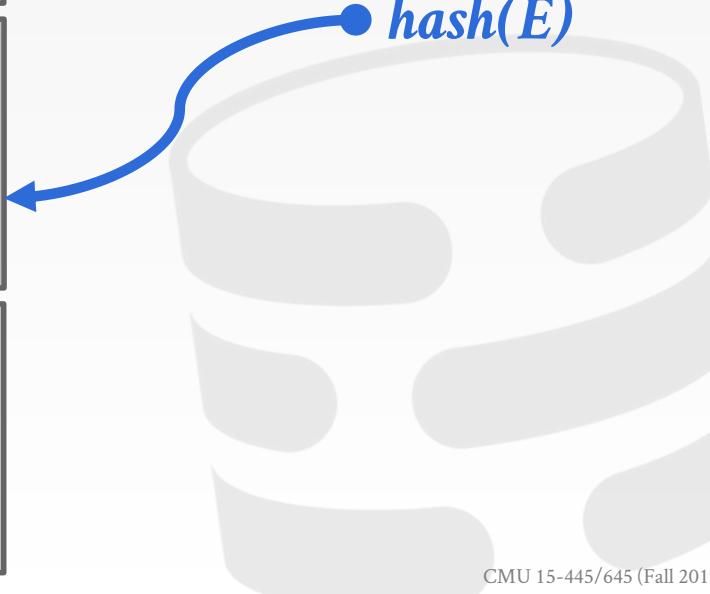


<i>B val</i>	0

<i>A val</i>	1
<i>C val</i>	1

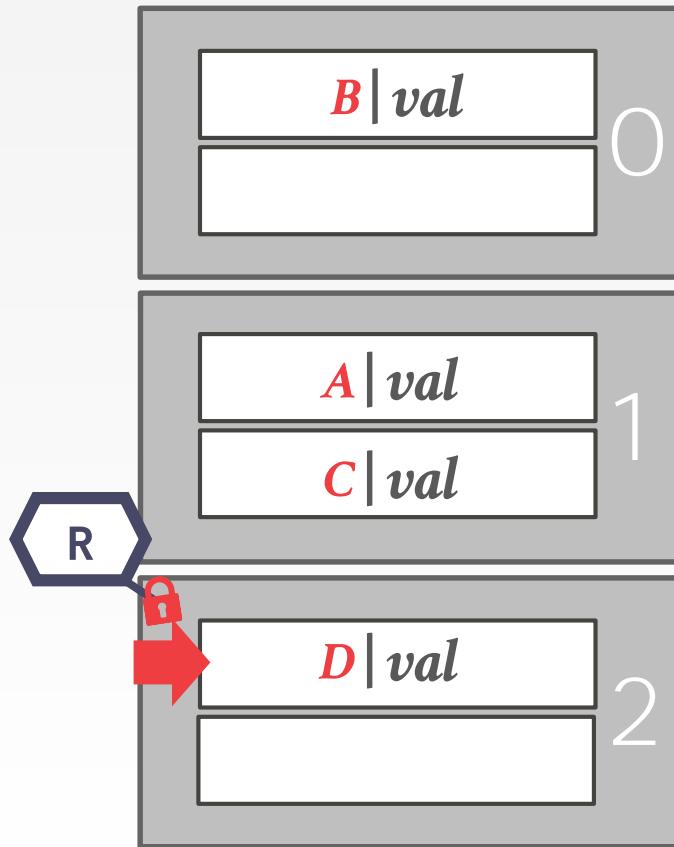
<i>D val</i>	2

T₂: Insert E
 $\text{hash}(E)$



HASH TABLE – PAGE LATCHES

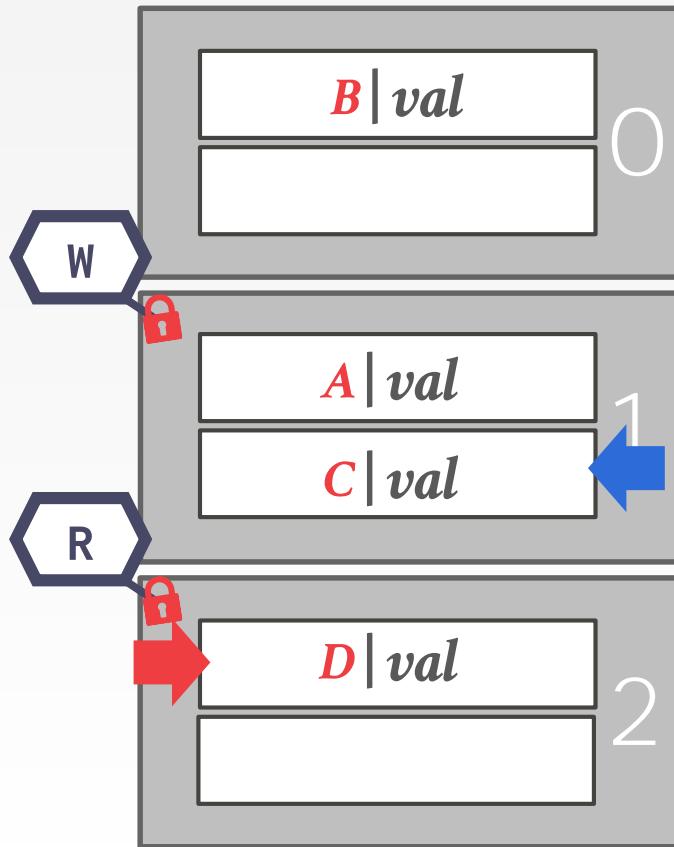
T₁: Find D
 $hash(D)$



T₂: Insert E
 $hash(E)$

HASH TABLE – PAGE LATCHES

T₁: Find D
 $hash(D)$

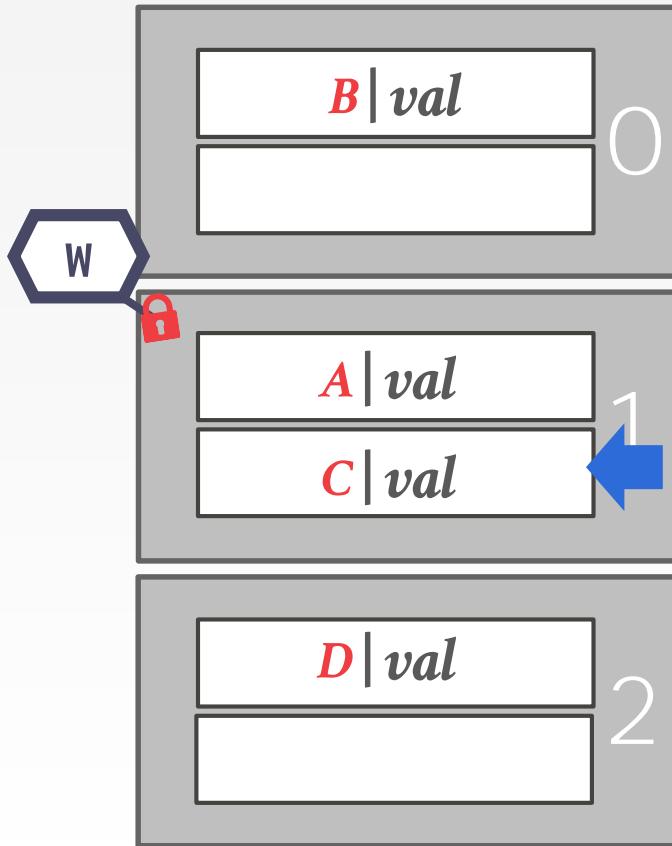


T₂: Insert E
 $hash(E)$



HASH TABLE – PAGE LATCHES

T₁: Find D
 $hash(D)$

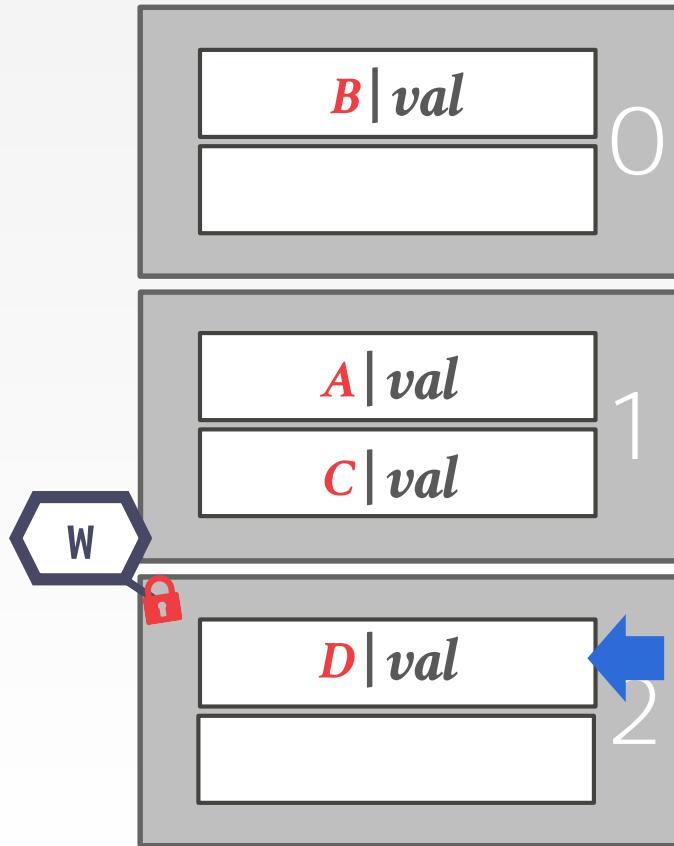


T₂: Insert E
 $hash(E)$



HASH TABLE – PAGE LATCHES

T₁: Find D
hash(D)

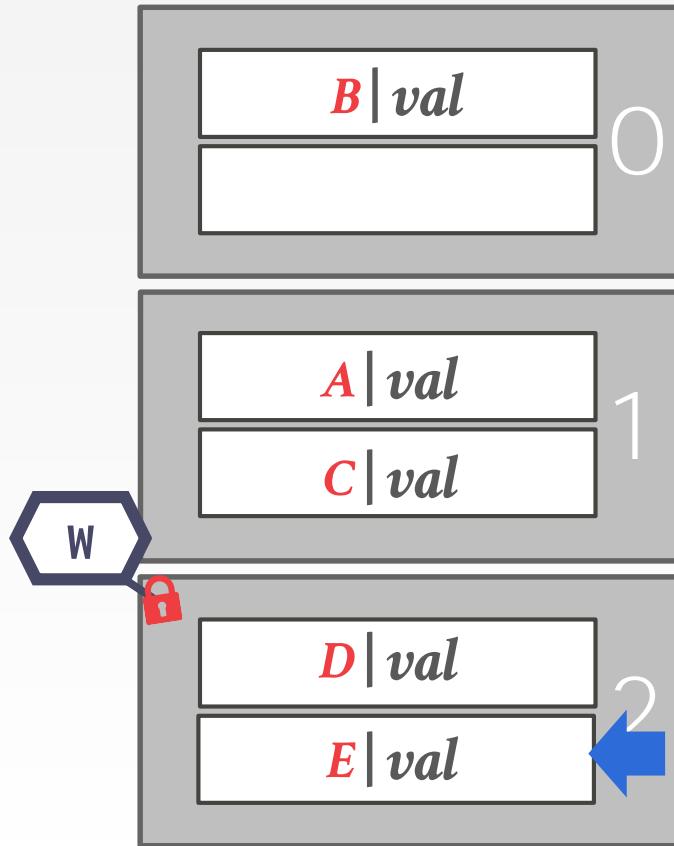


T₂: Insert E
hash(E)



HASH TABLE – PAGE LATCHES

T₁: Find D
hash(D)

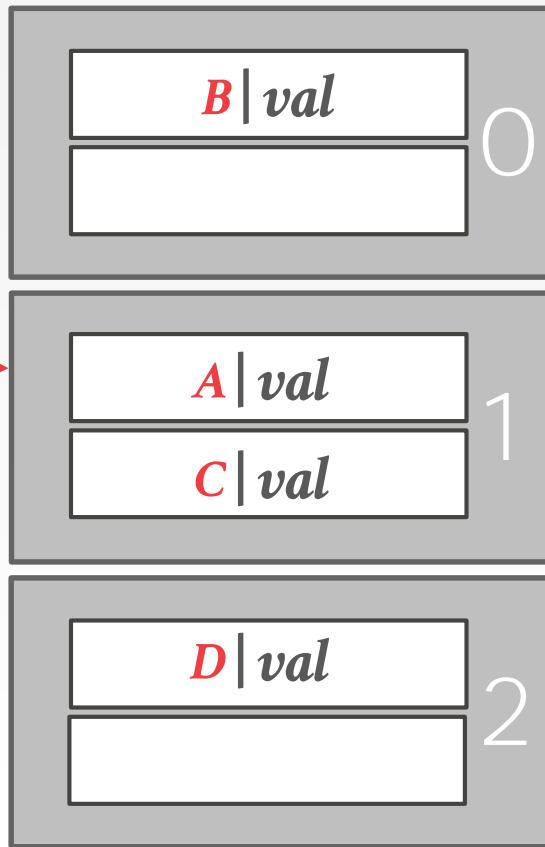


T₂: Insert E
hash(E)



HASH TABLE – SLOT LATCHES

T₁: Find D
 $hash(D)$

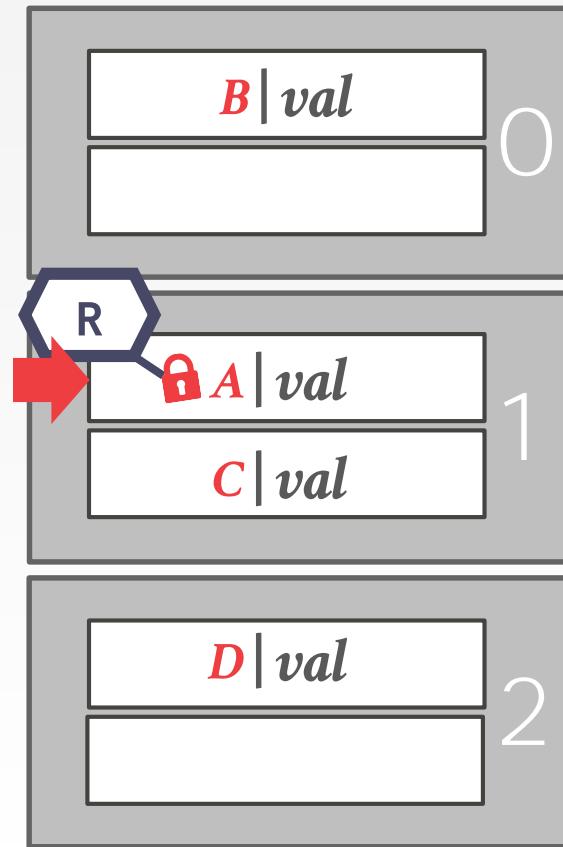


T₂: Insert E
 $hash(E)$



HASH TABLE – SLOT LATCHES

T₁: Find D
 $hash(D)$

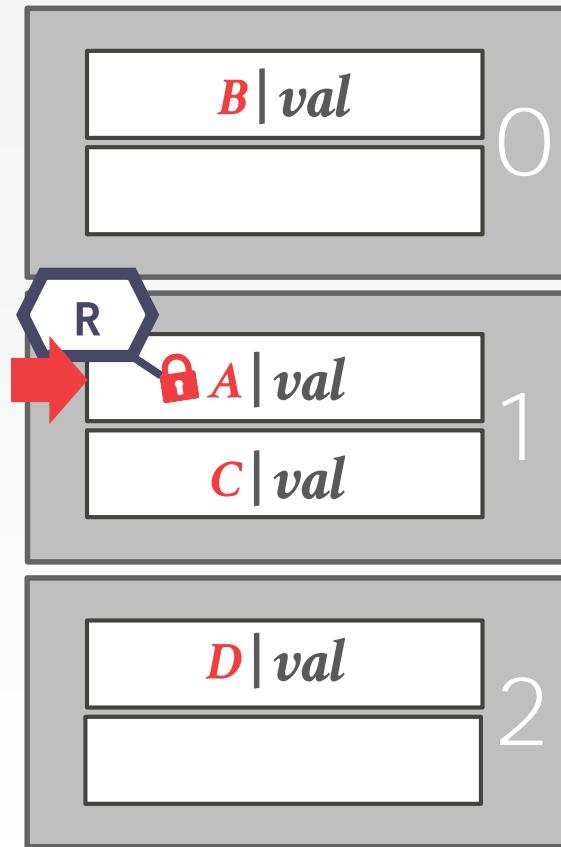


T₂: Insert E
 $hash(E)$

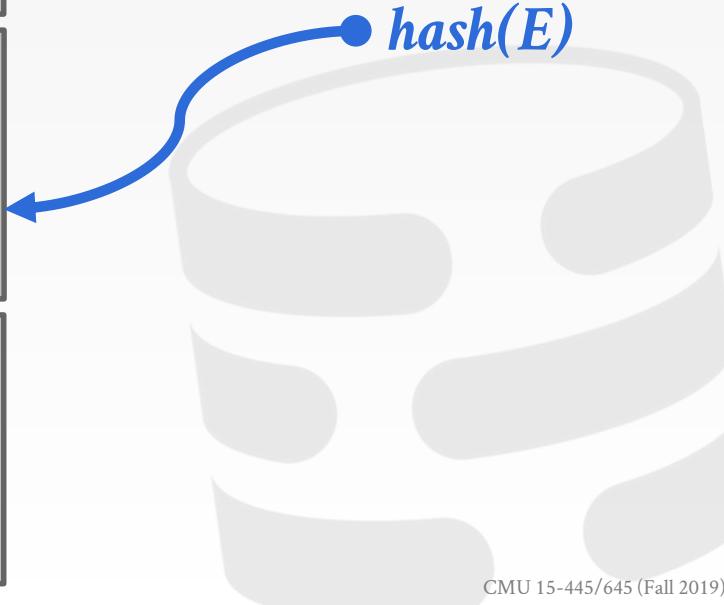


HASH TABLE – SLOT LATCHES

T₁: Find D
 $hash(D)$



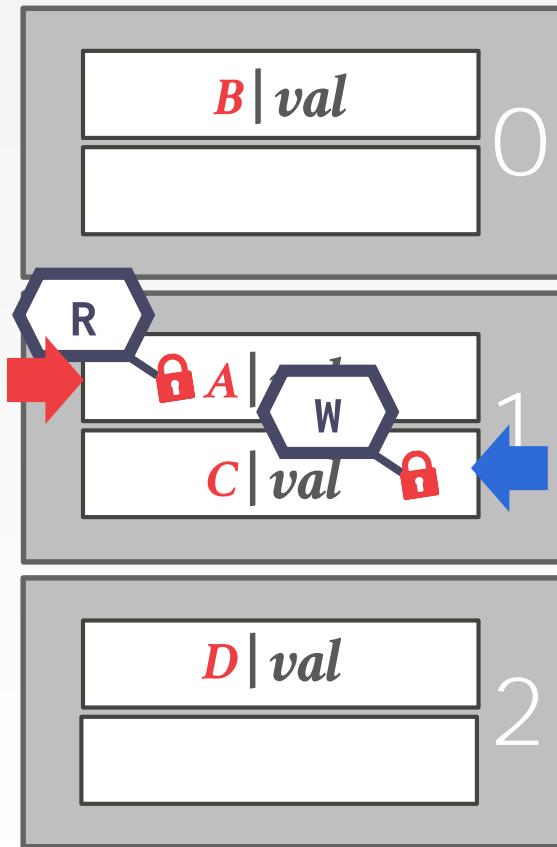
T₂: Insert E
 $hash(E)$



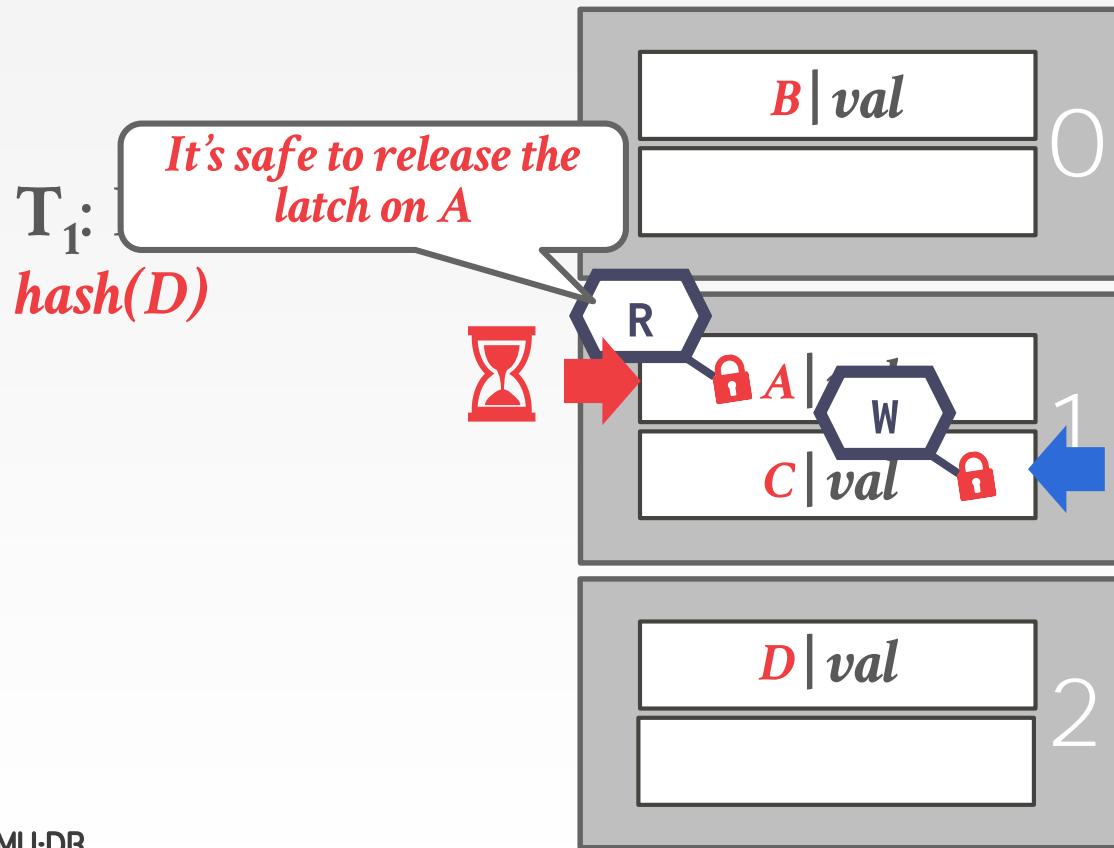
HASH TABLE – SLOT LATCHES

T₁: Find D
hash(D)

T₂: Insert E
hash(E)



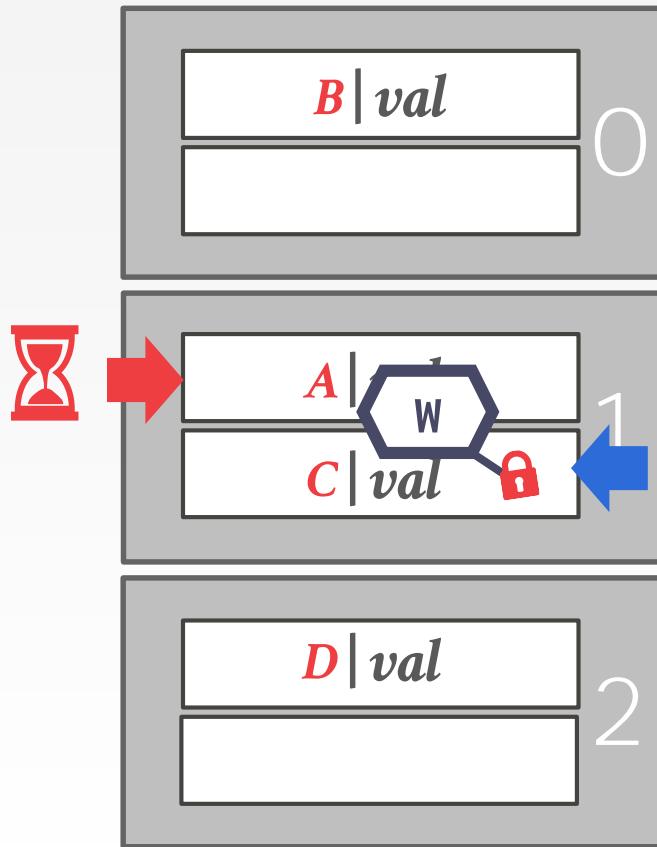
HASH TABLE – SLOT LATCHES



T_2 : Insert E
 $hash(E)$

HASH TABLE – SLOT LATCHES

T₁: Find D
hash(D)

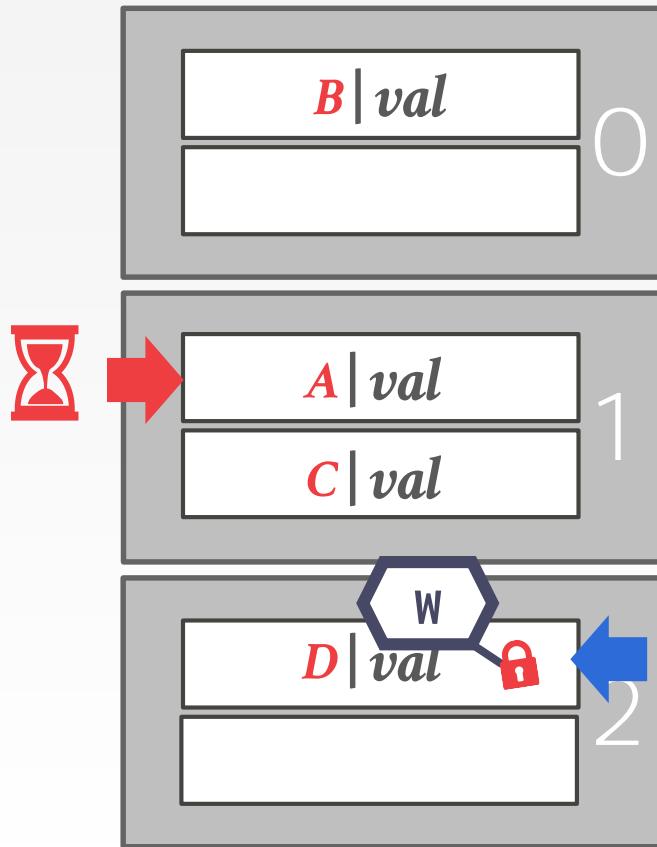


T₂: Insert E
hash(E)



HASH TABLE – SLOT LATCHES

T₁: Find D
hash(D)

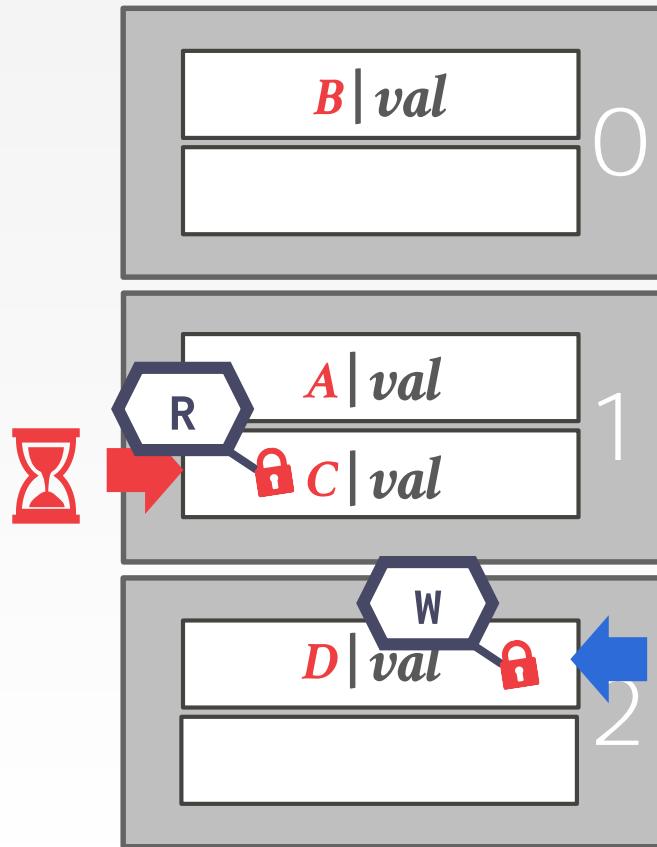


T₂: Insert E
hash(E)



HASH TABLE – SLOT LATCHES

T₁: Find D
hash(D)

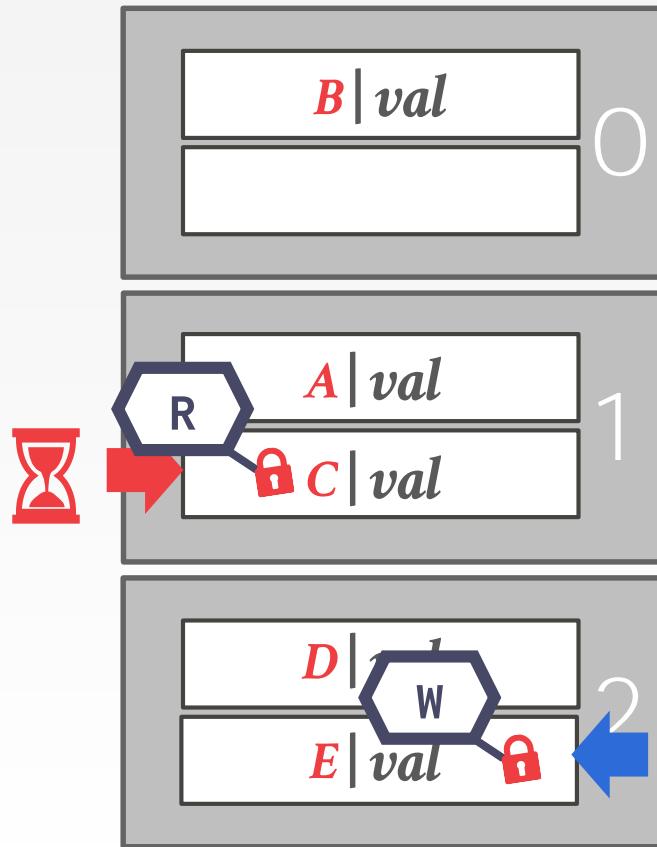


T₂: Insert E
hash(E)



HASH TABLE – SLOT LATCHES

T_1 : Find D
 $\text{hash}(D)$

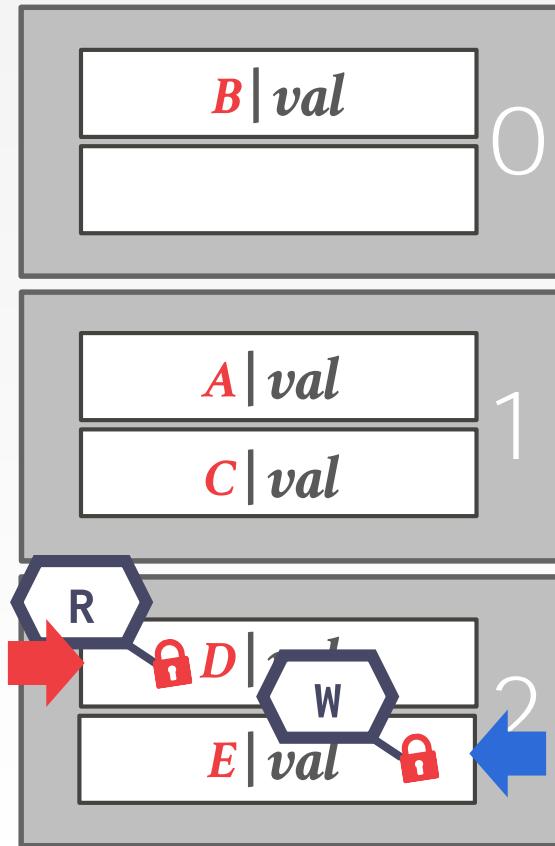


T_2 : Insert E
 $\text{hash}(E)$



HASH TABLE – SLOT LATCHES

T₁: Find D
hash(D)



T₂: Insert E
hash(E)



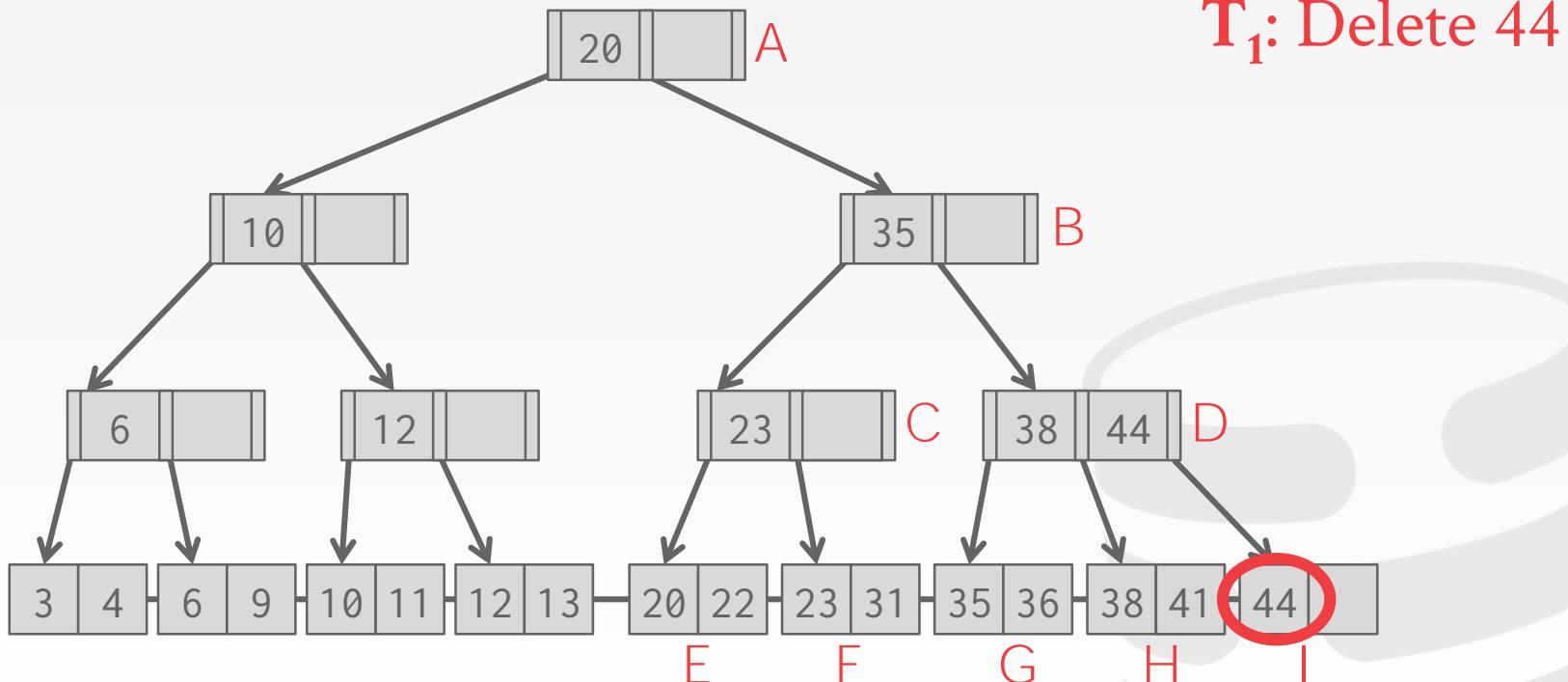
B+TREE CONCURRENCY CONTROL

We want to allow multiple threads to read and update a B+Tree at the same time.

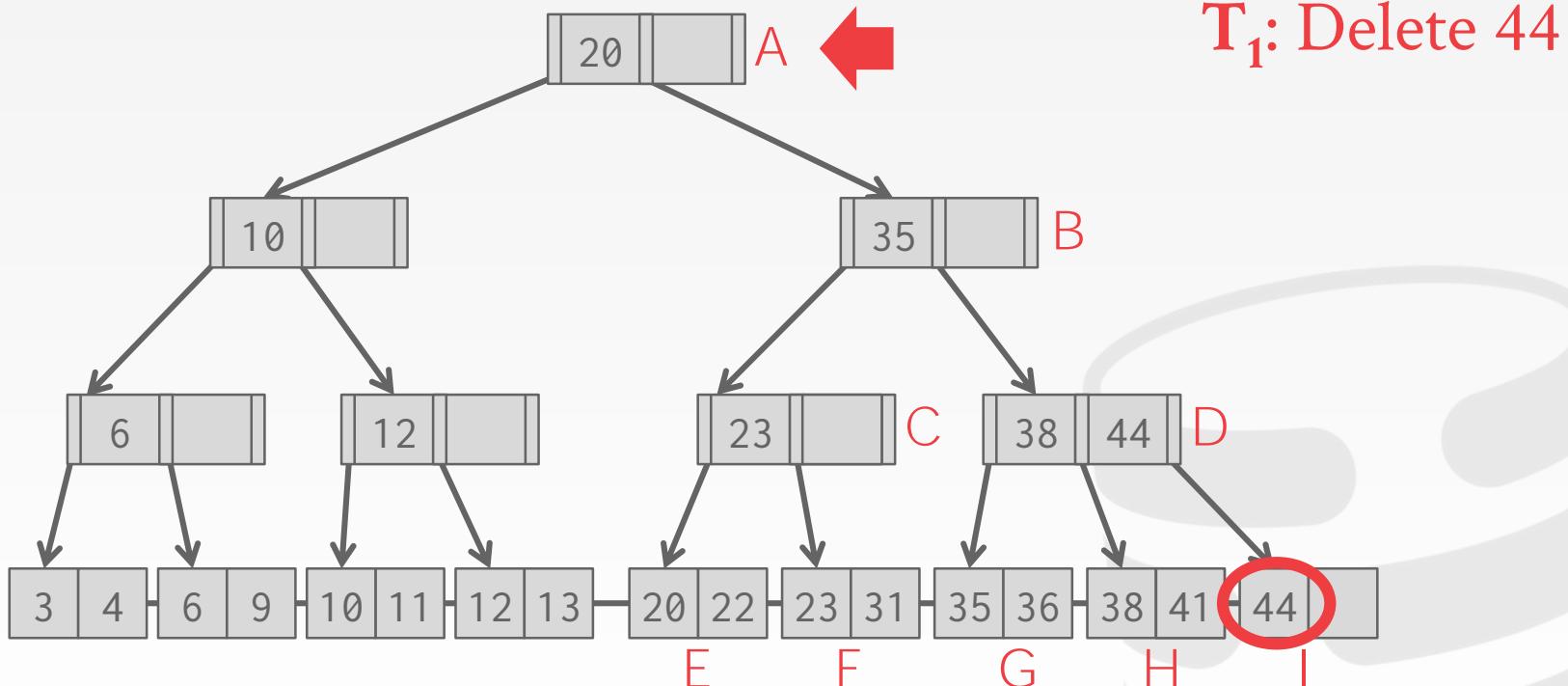
We need to protect from two types of problems:

- Threads trying to modify the contents of a node at the same time.
- One thread traversing the tree while another thread splits/merges nodes.

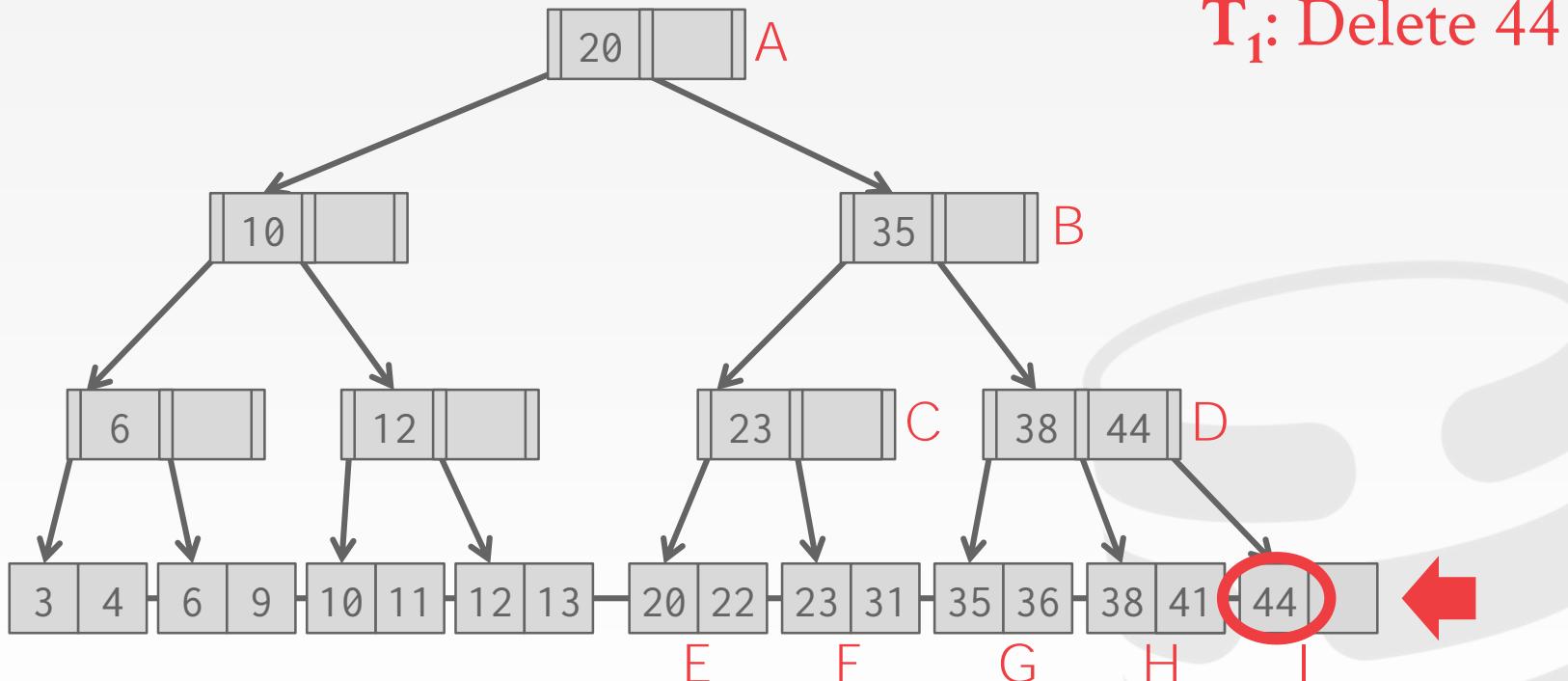
B+TREE MULTI-THREADED EXAMPLE



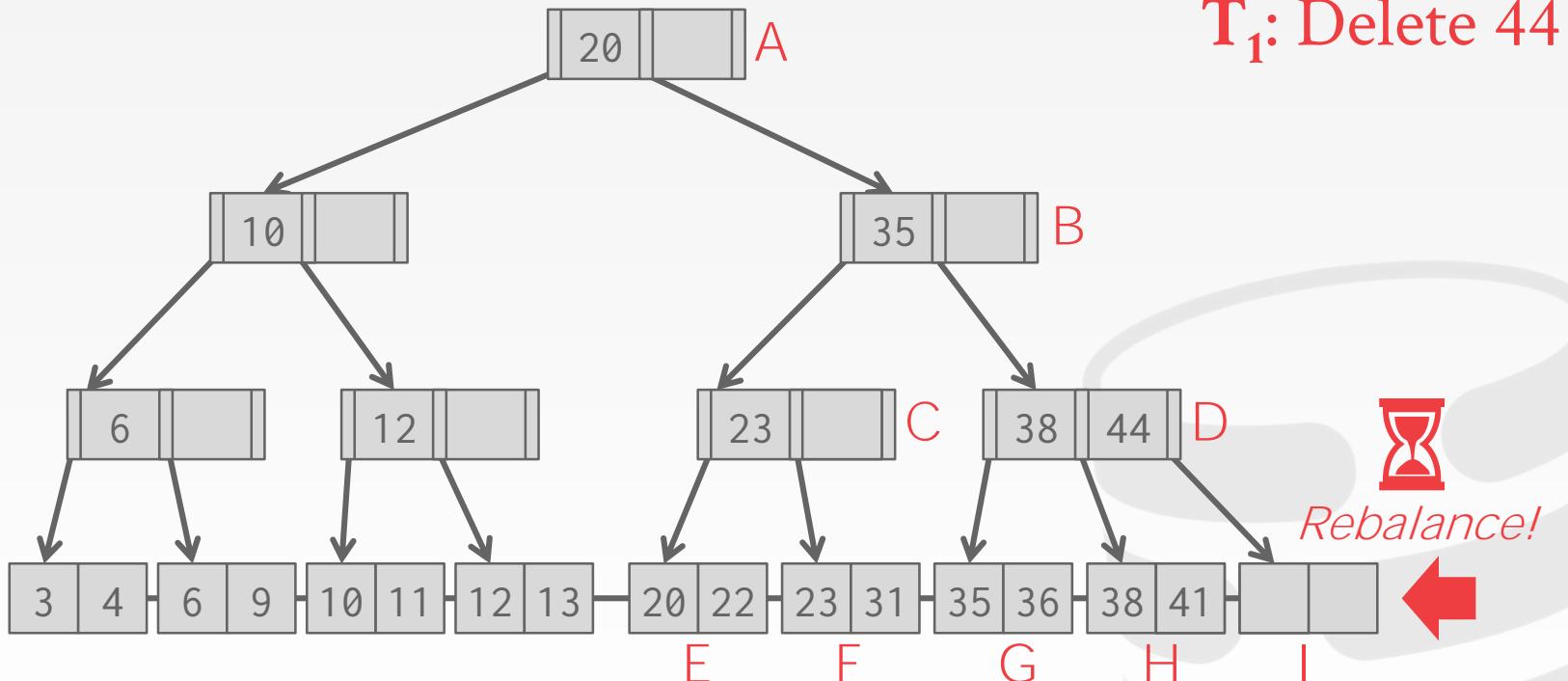
B+TREE MULTI-THREADED EXAMPLE



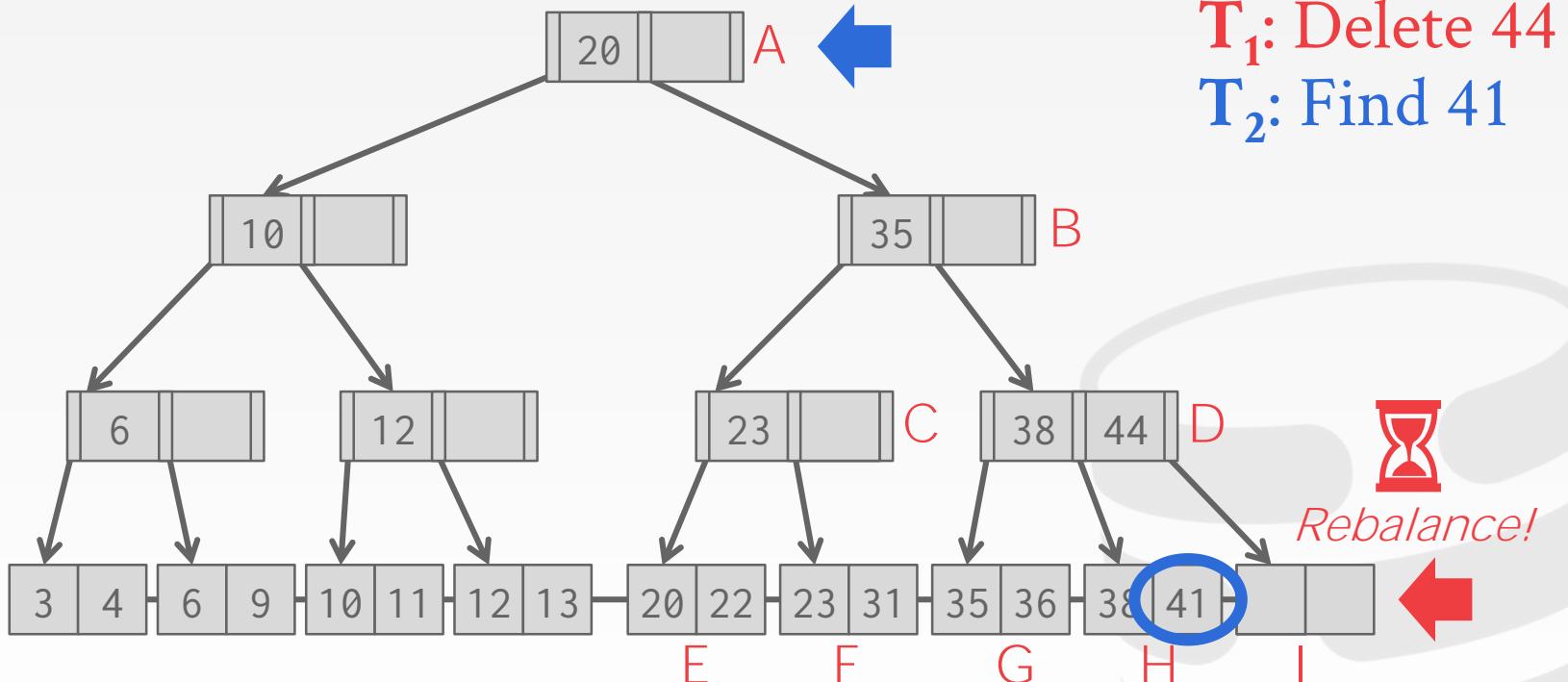
B+TREE MULTI-THREADED EXAMPLE



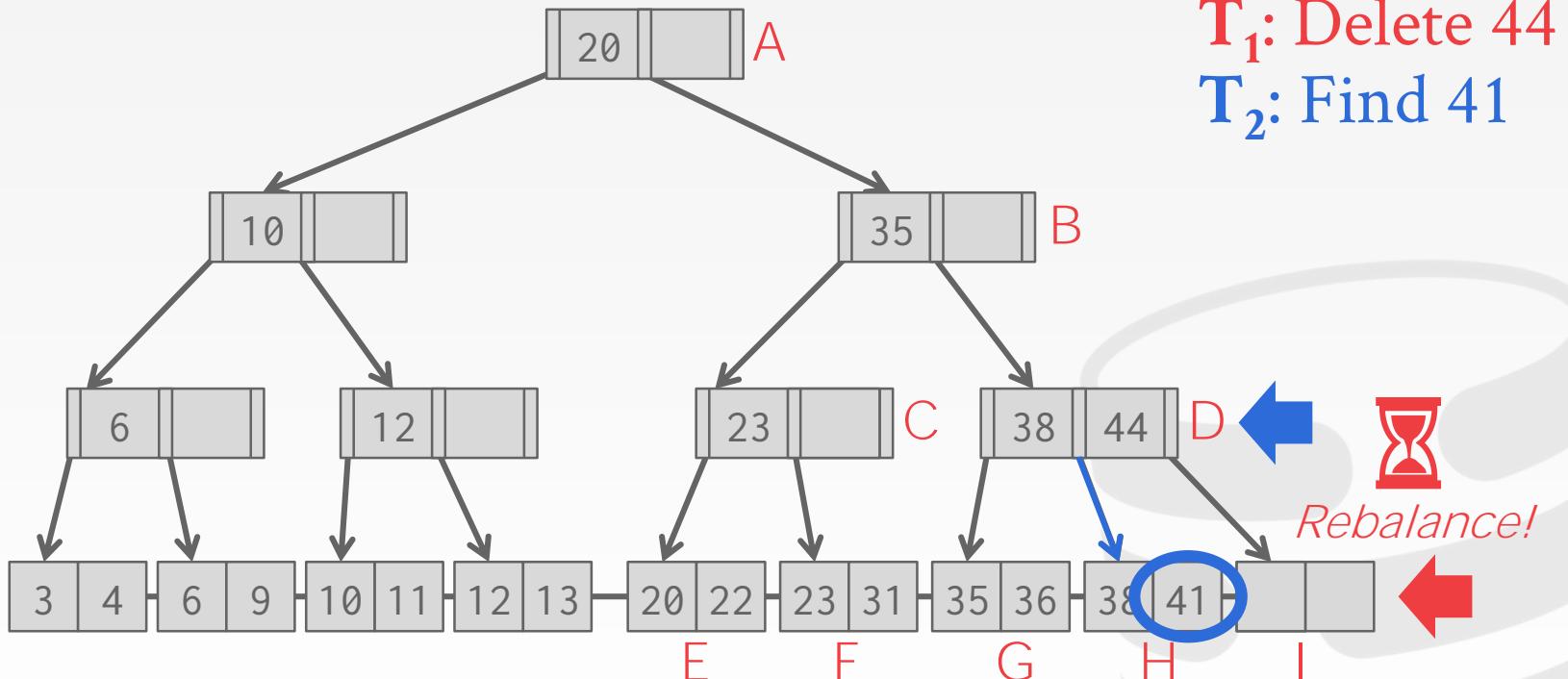
B+TREE MULTI-THREADED EXAMPLE



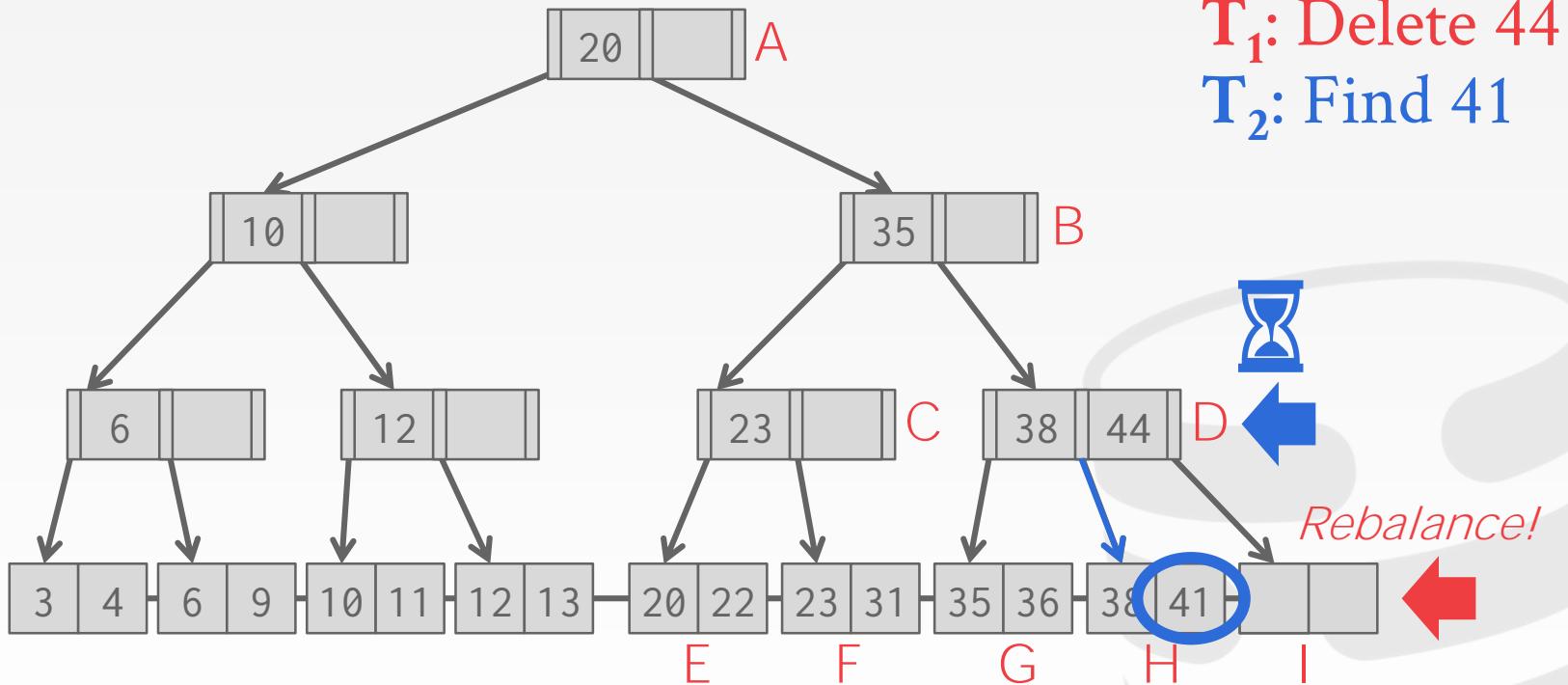
B+TREE MULTI-THREADED EXAMPLE



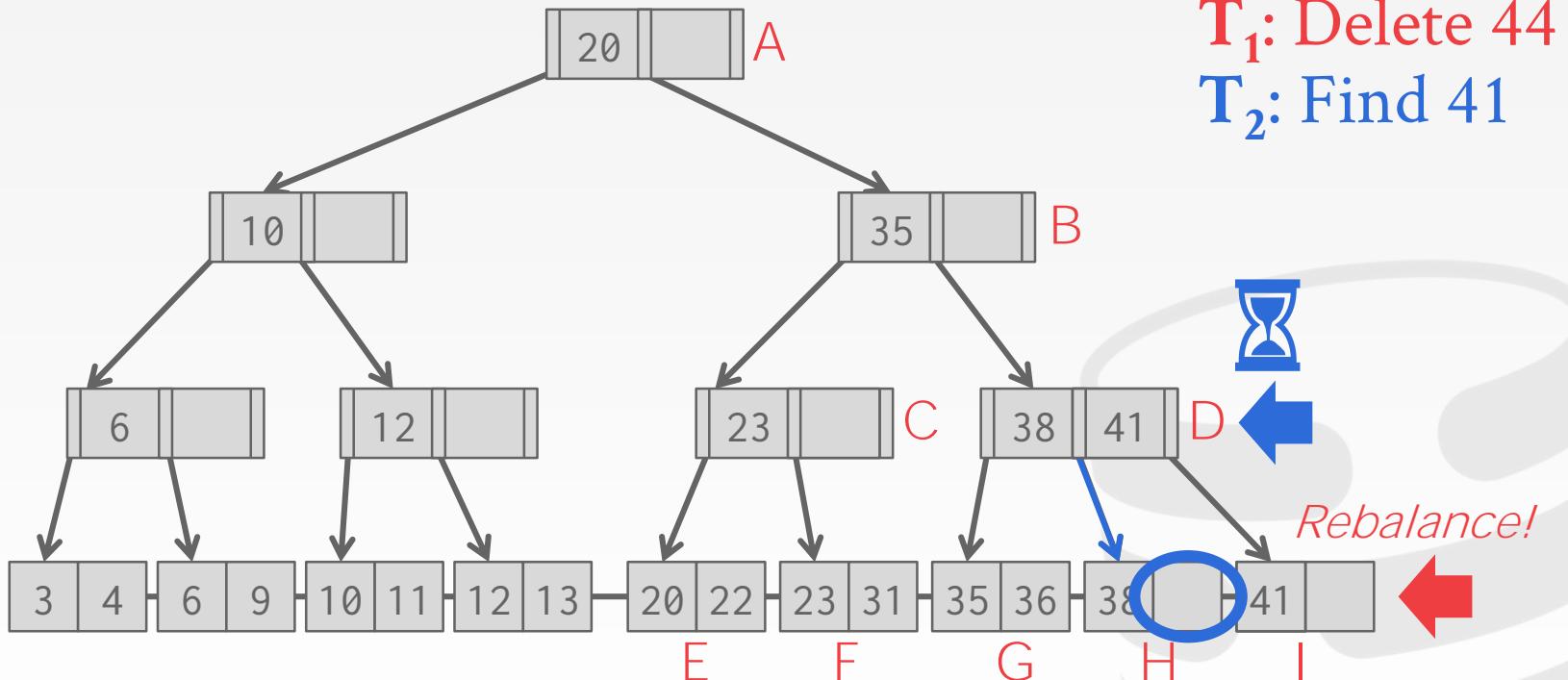
B+TREE MULTI-THREADED EXAMPLE



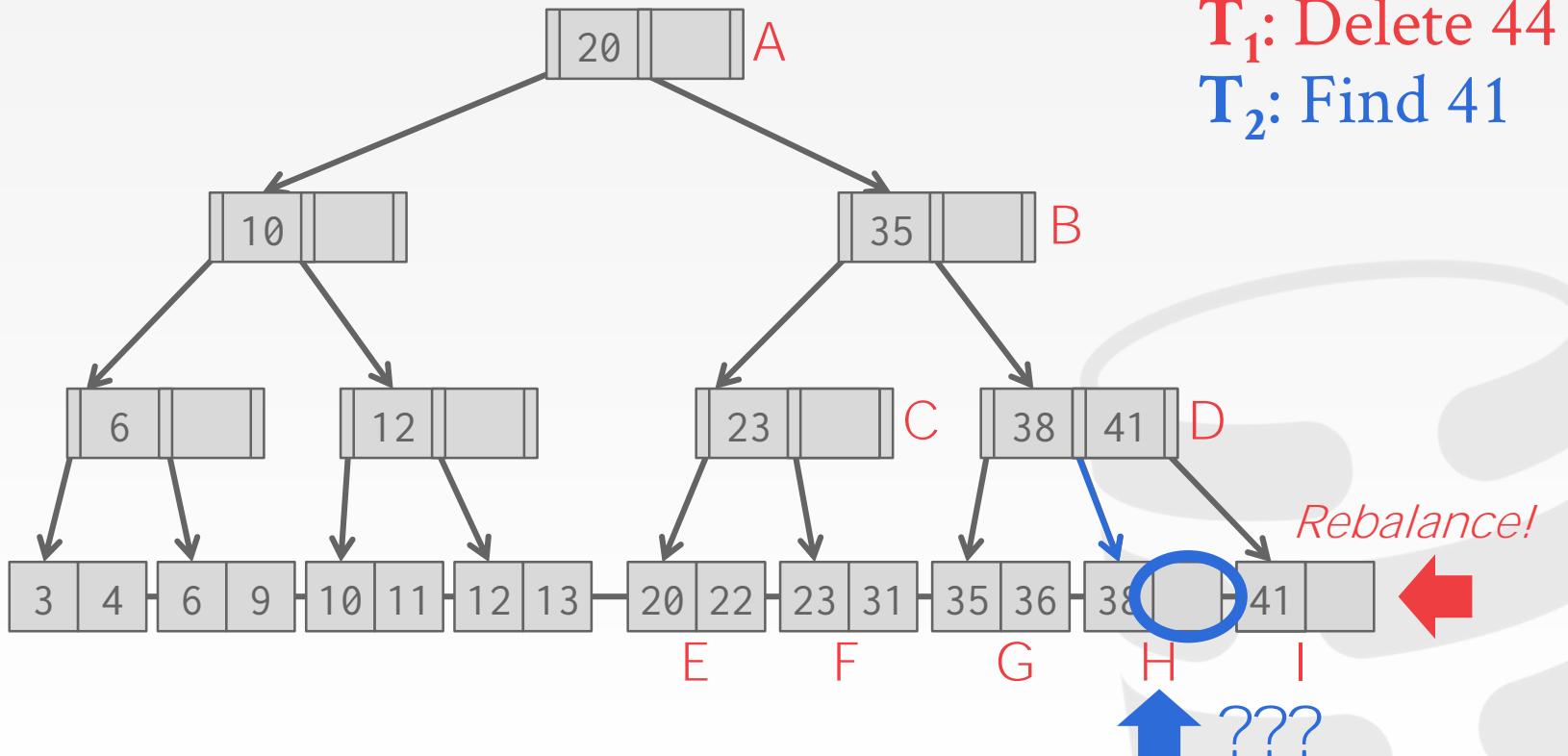
B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE



LATCH CRABBING/COUPLING

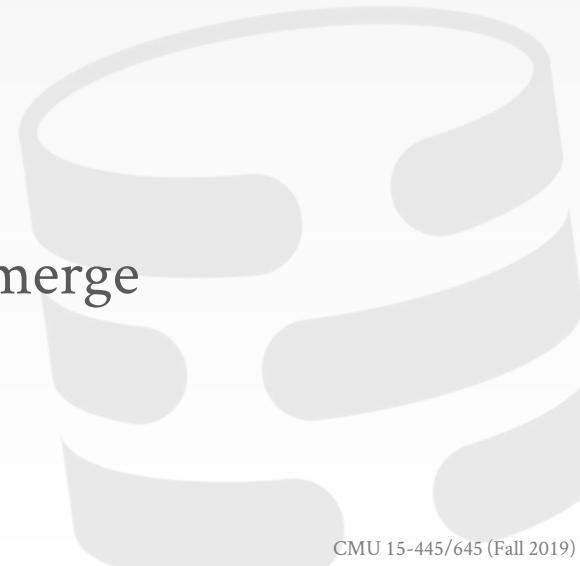
Protocol to allow multiple threads to access/modify B+Tree at the same time.

Basic Idea:

- Get latch for parent.
- Get latch for child
- Release latch for parent if “safe”.

A **safe node** is one that will not split or merge when updated.

- Not full (on insertion)
- More than half-full (on deletion)



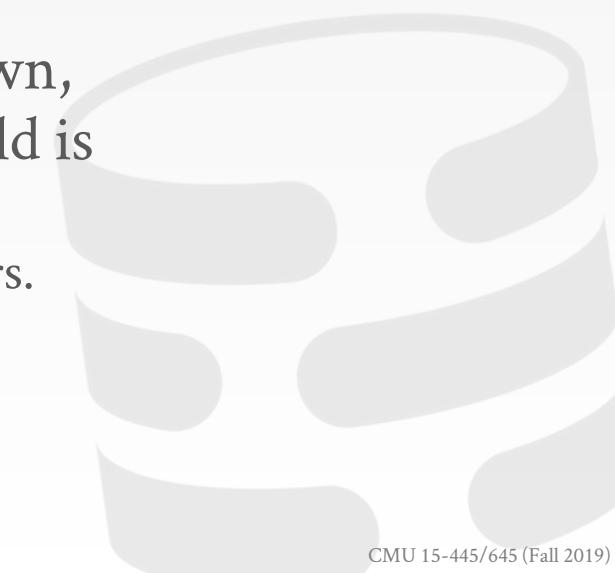
LATCH CRABBING/COUPLING

Find: Start at root and go down; repeatedly,

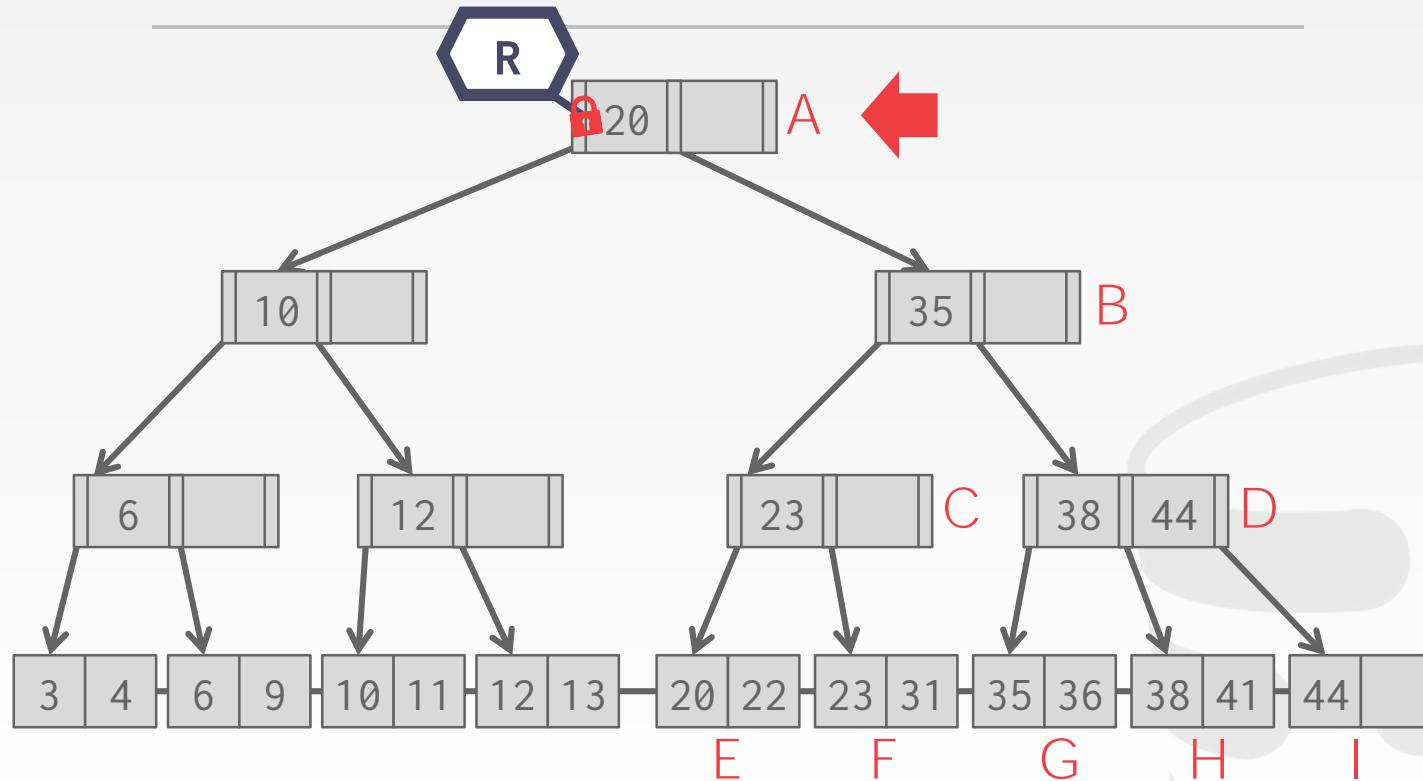
- Acquire **R** latch on child
- Then unlatch parent

Insert/Delete: Start at root and go down,
obtaining **W** latches as needed. Once child is
latched, check if it is safe:

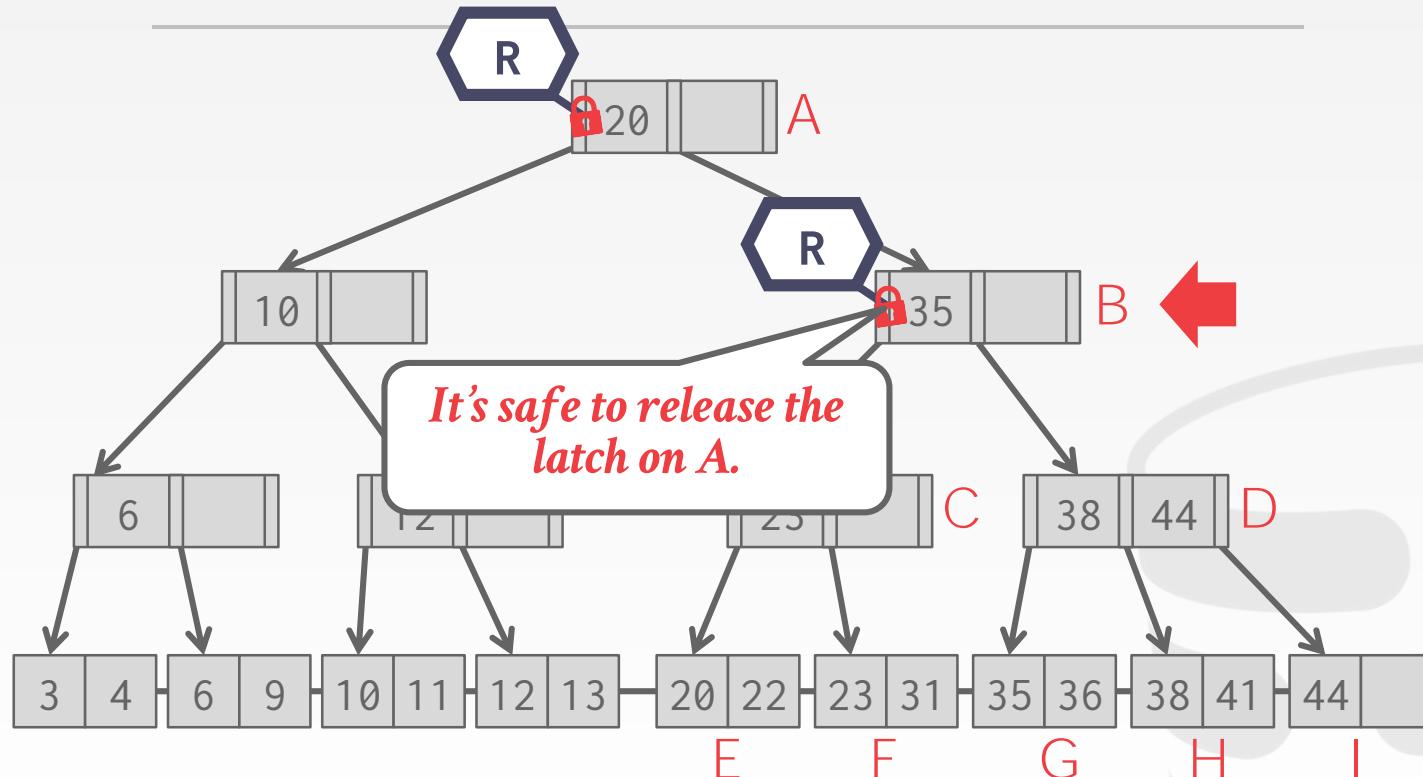
- If child is safe, release all latches on ancestors.



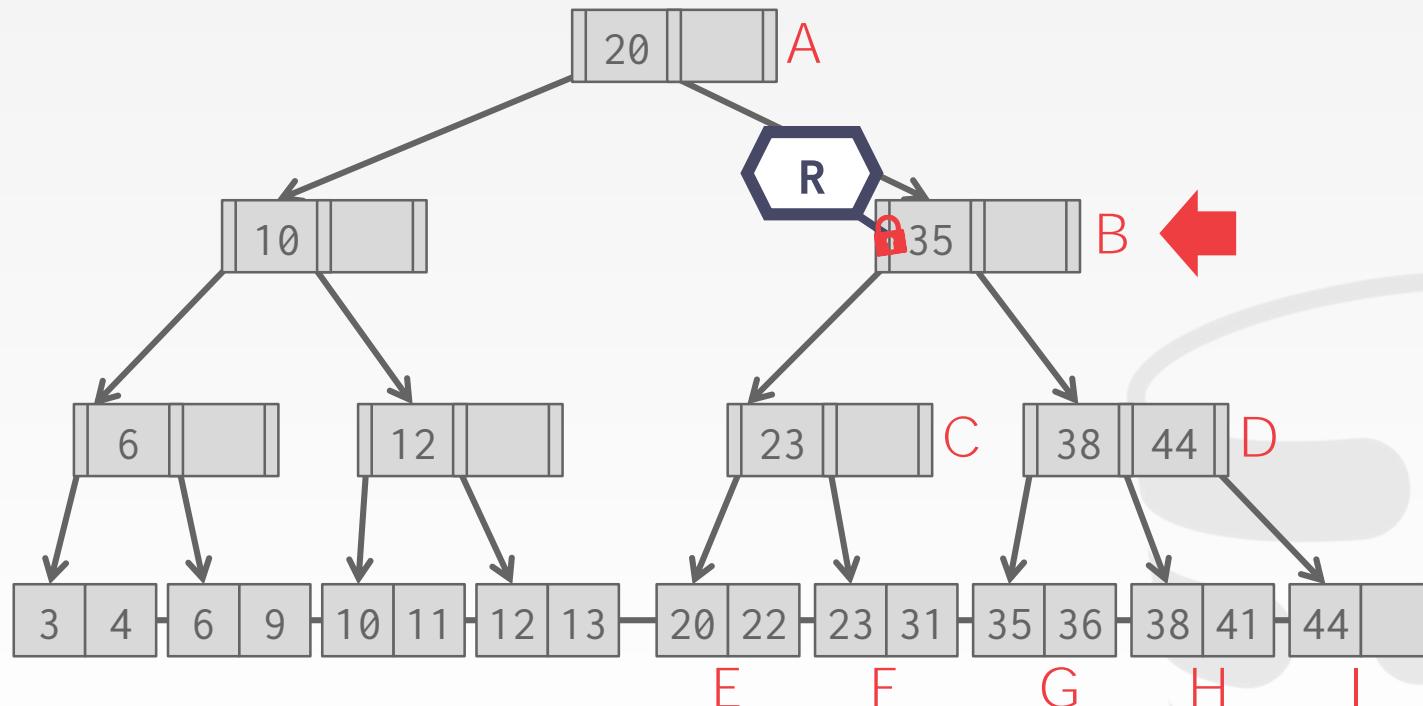
EXAMPLE #1 – FIND 38



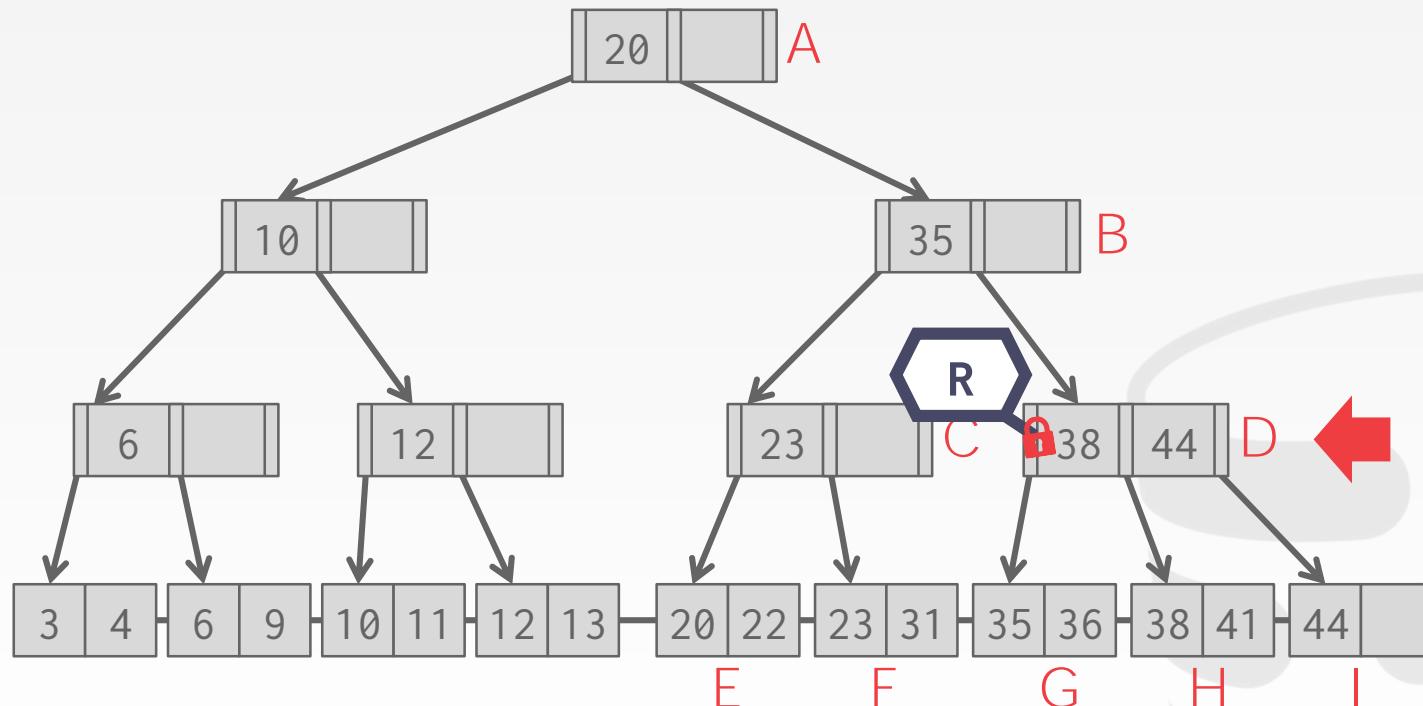
EXAMPLE #1 – FIND 38



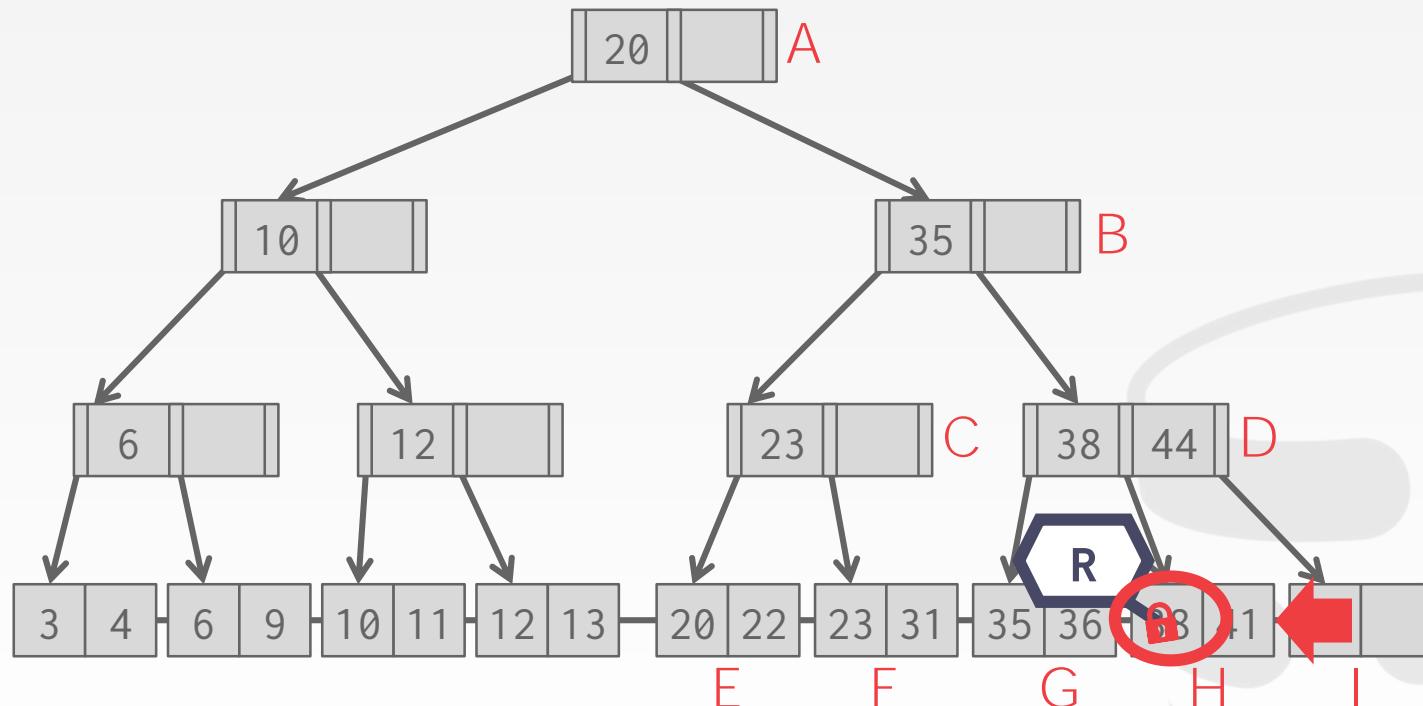
EXAMPLE #1 – FIND 38



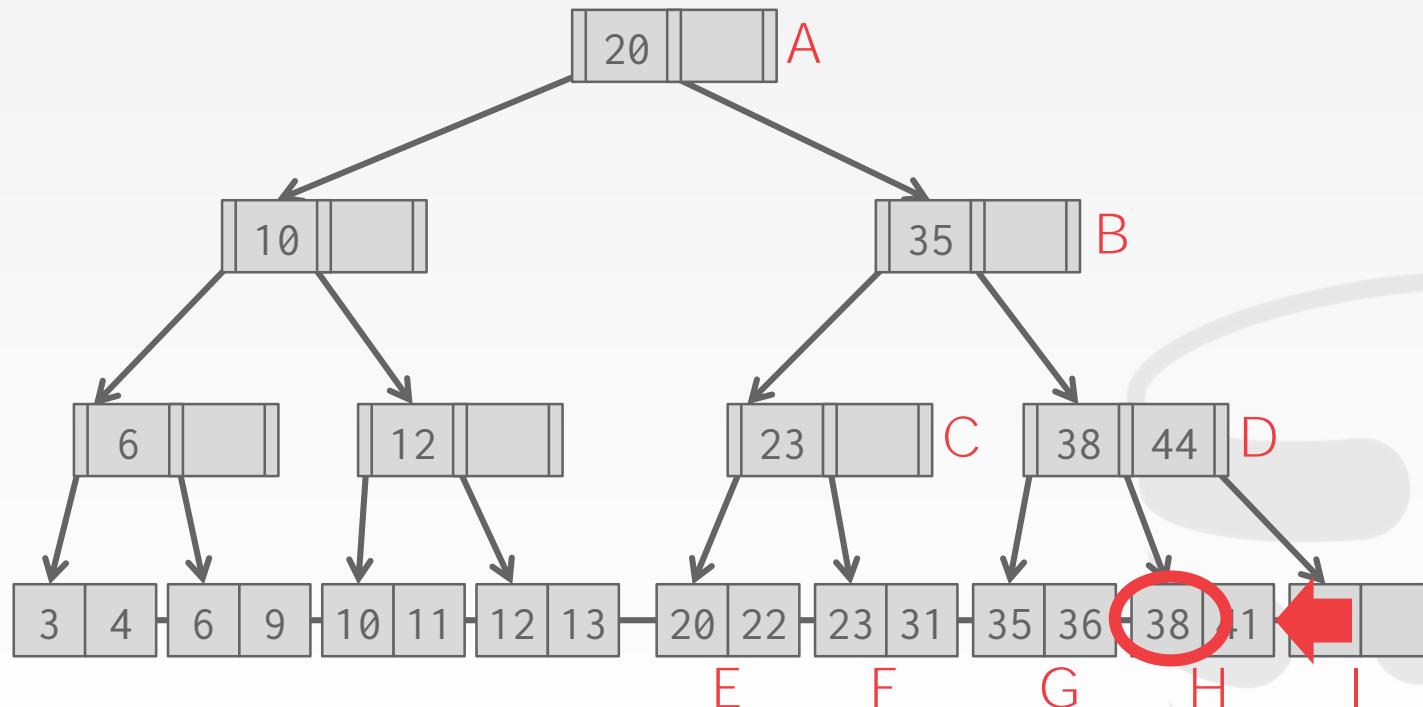
EXAMPLE #1 – FIND 38



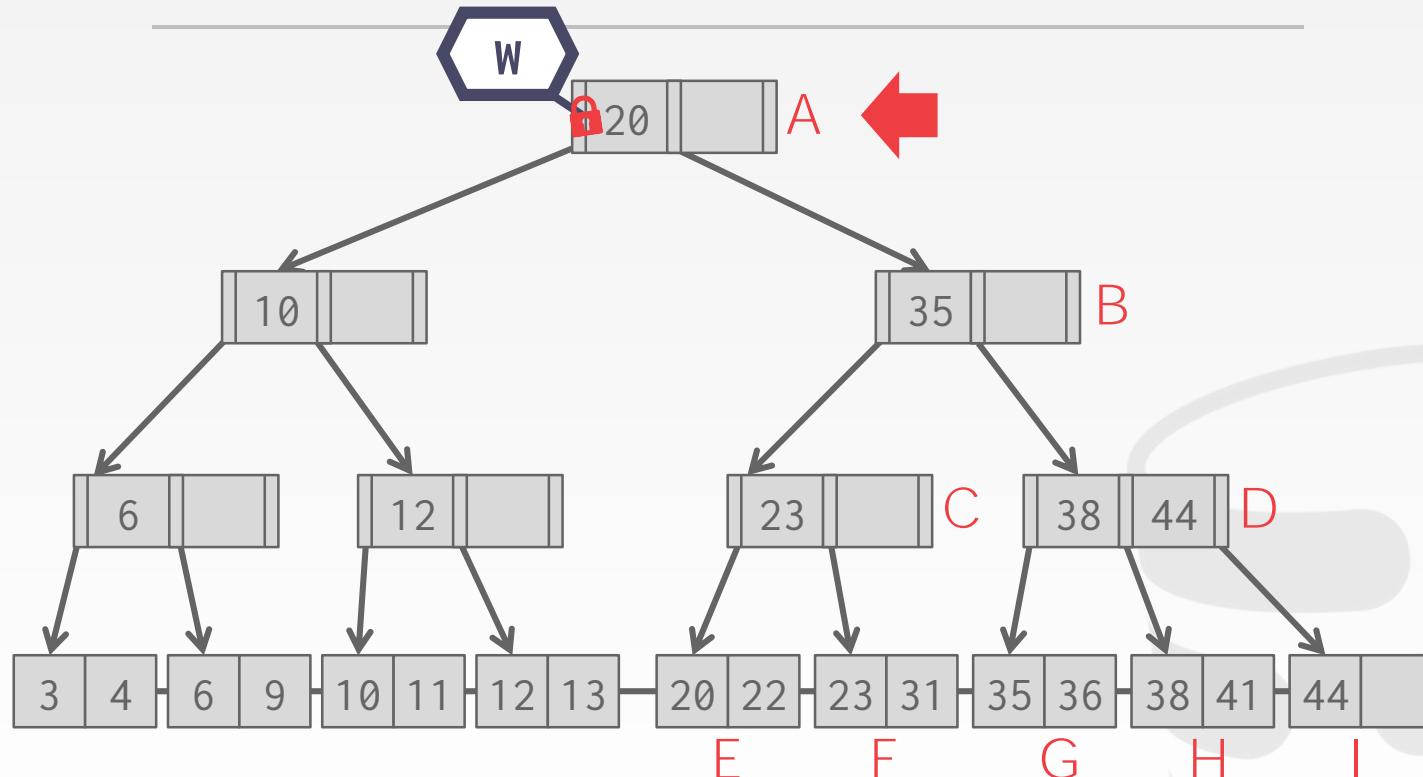
EXAMPLE #1 – FIND 38



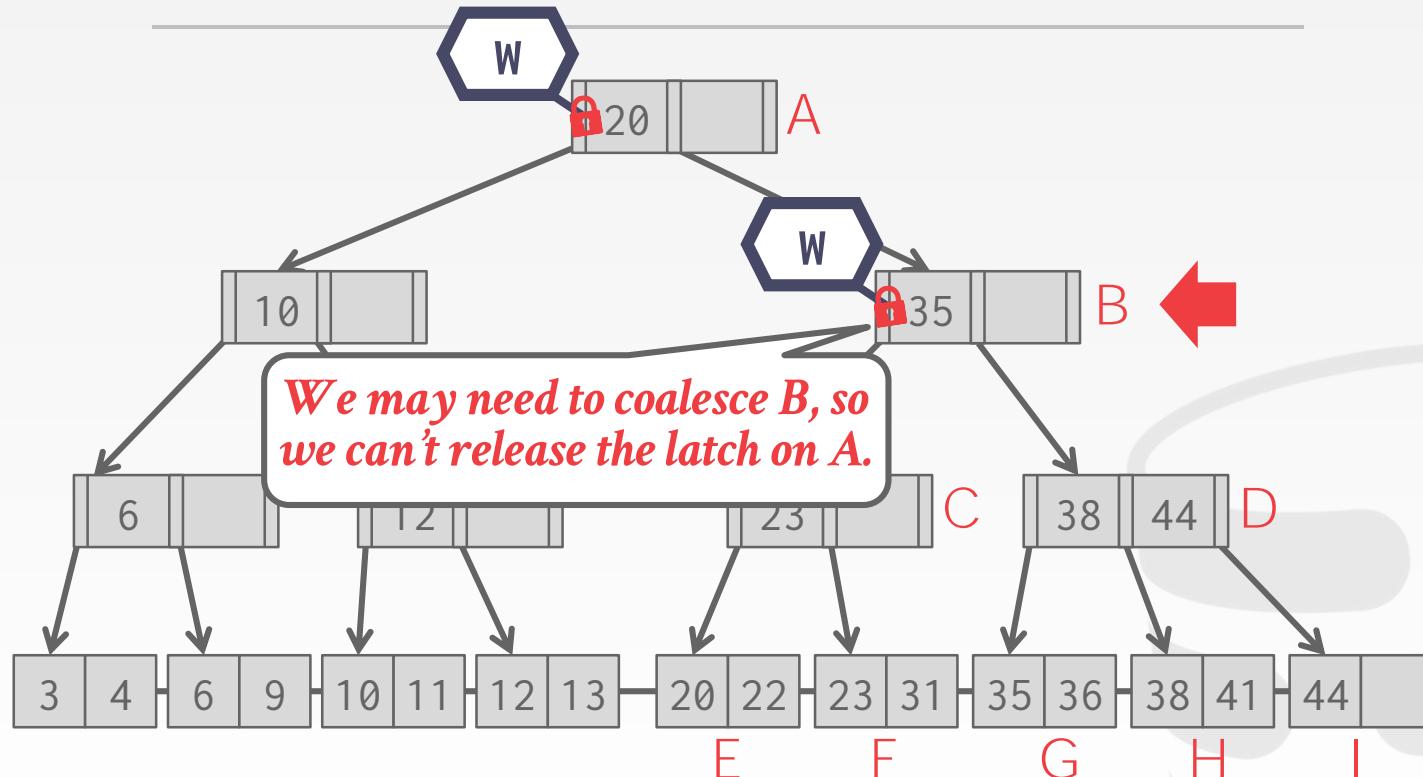
EXAMPLE #1 – FIND 38



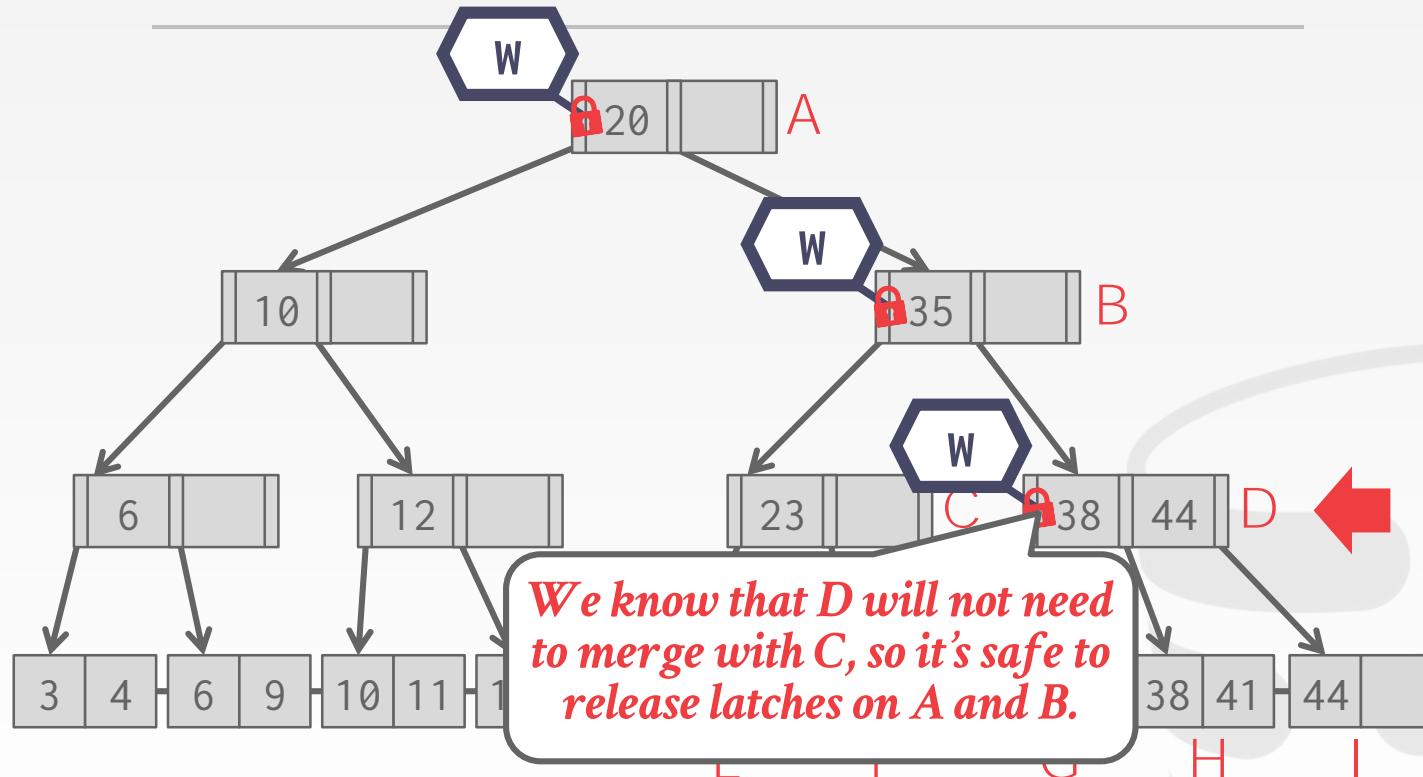
EXAMPLE #2 – DELETE 38



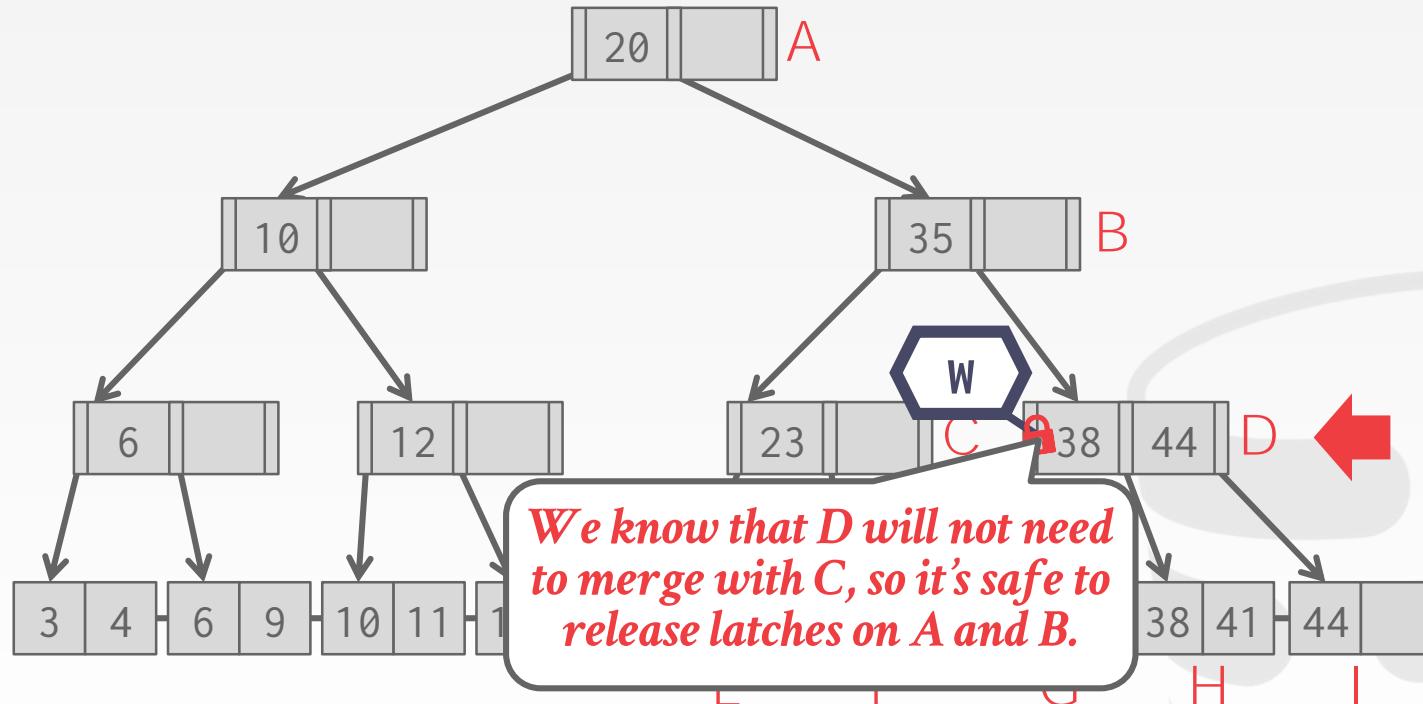
EXAMPLE #2 – DELETE 38



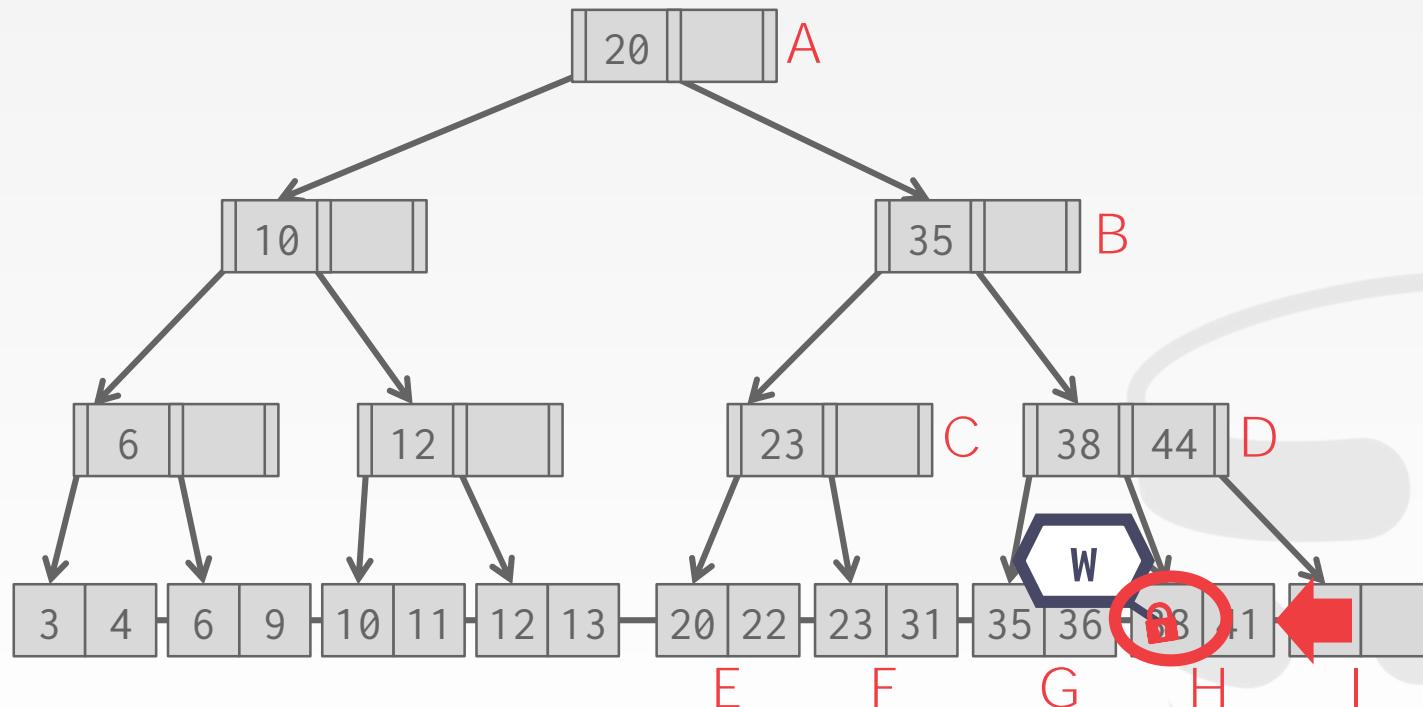
EXAMPLE #2 – DELETE 38



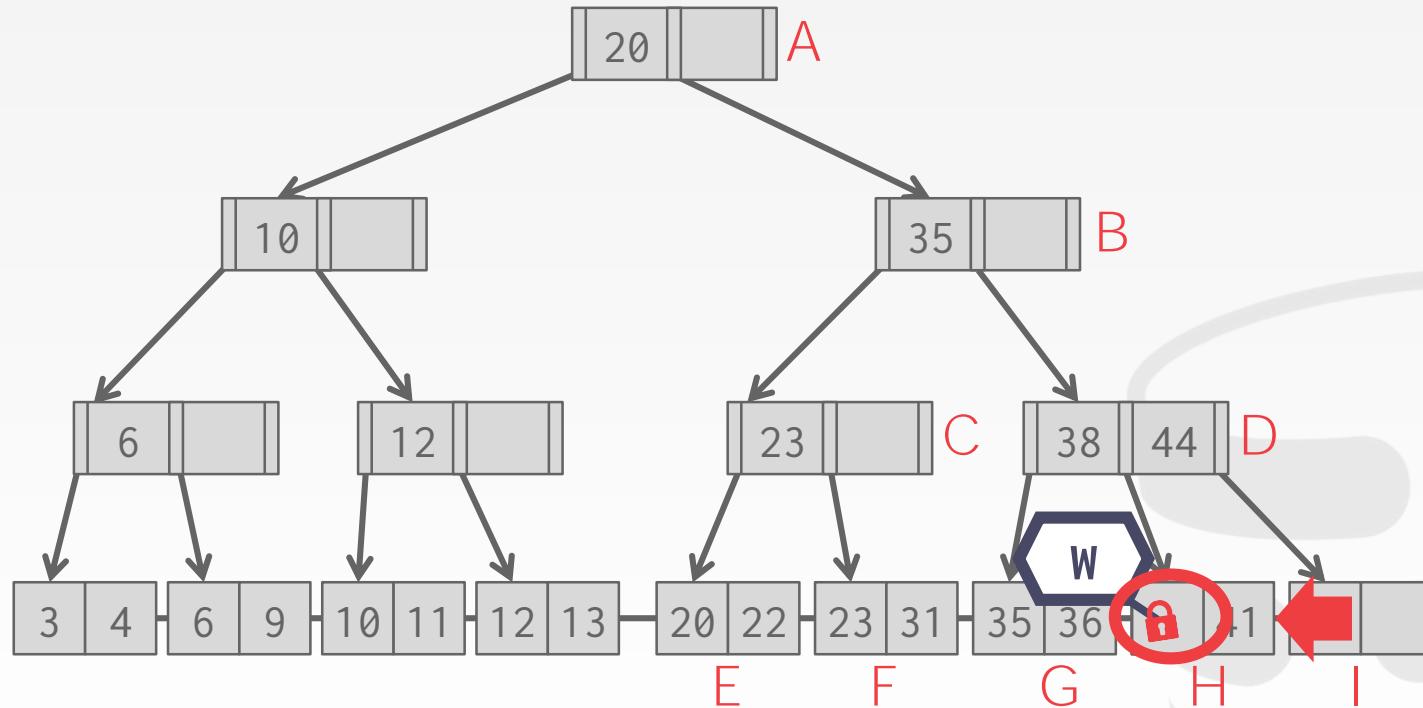
EXAMPLE #2 – DELETE 38



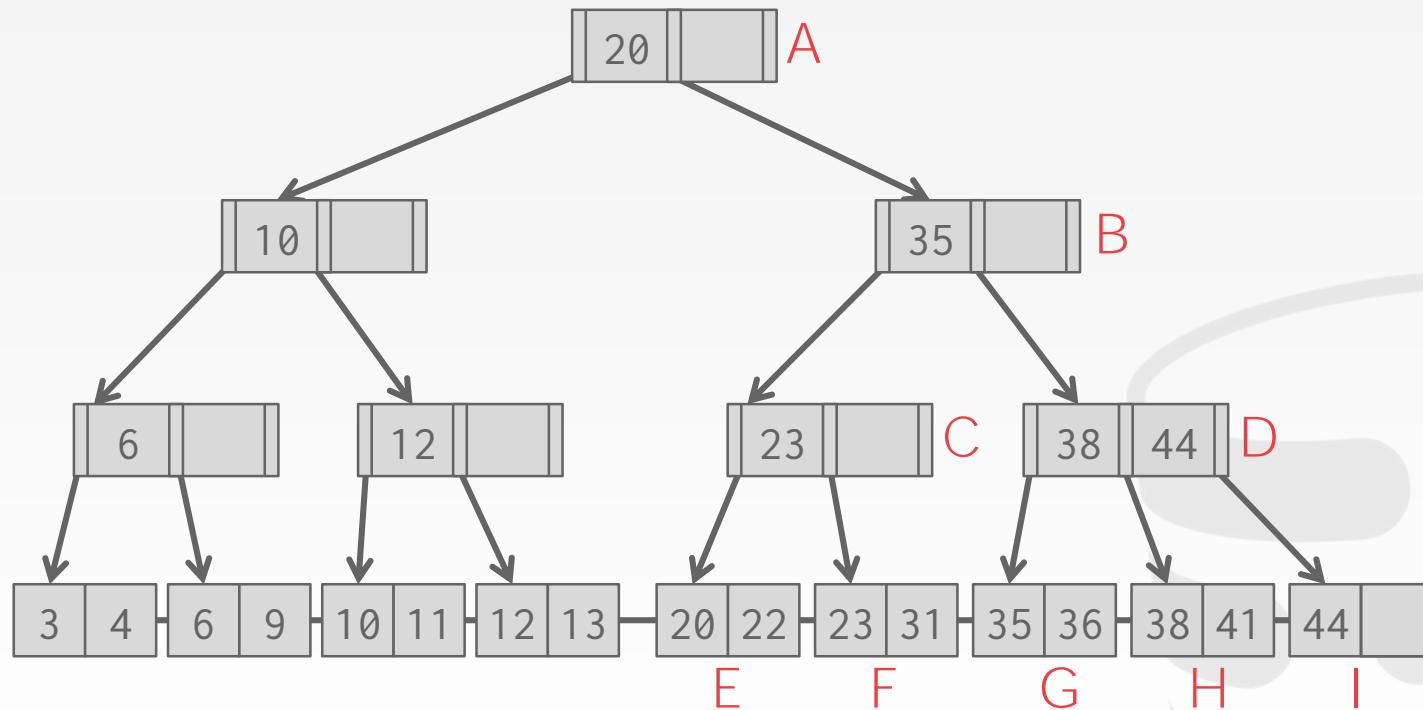
EXAMPLE #2 – DELETE 38



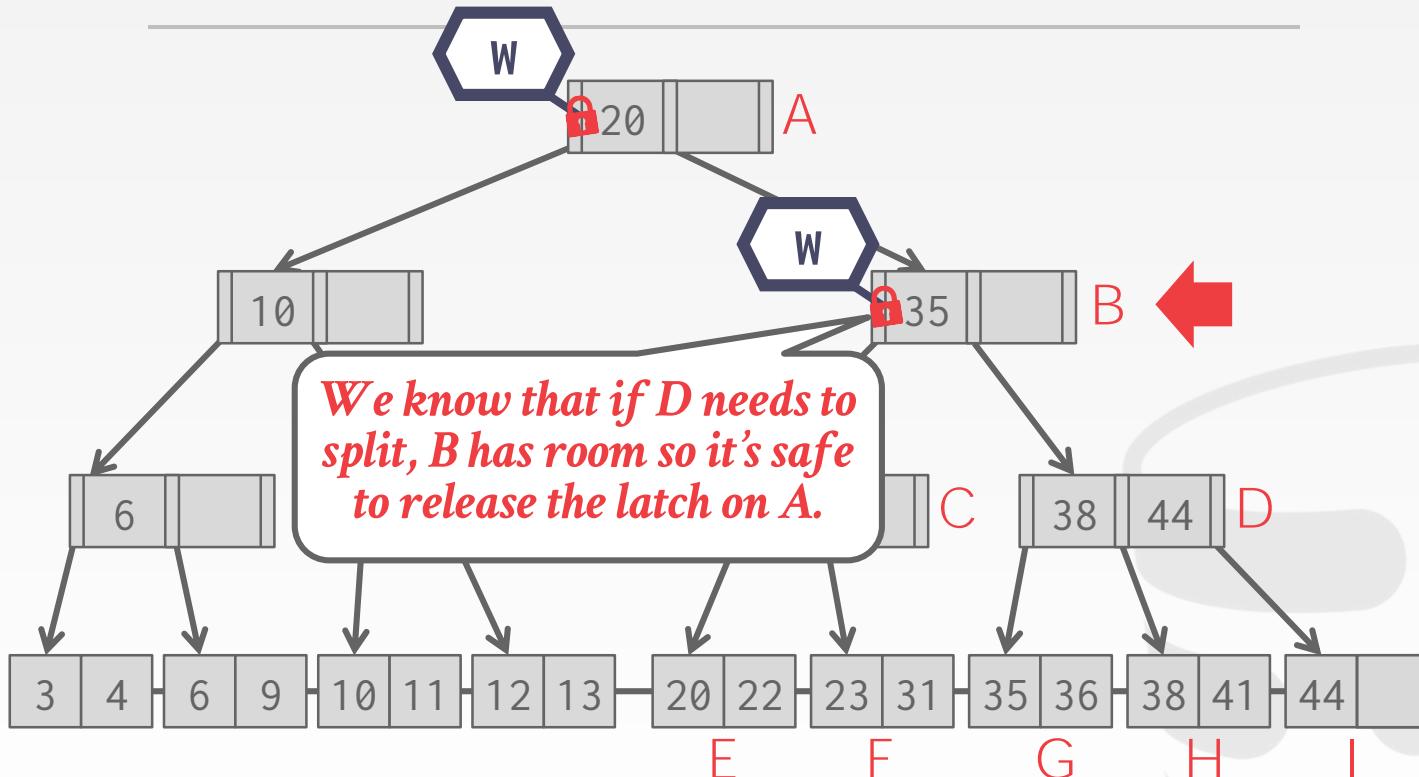
EXAMPLE #2 – DELETE 38



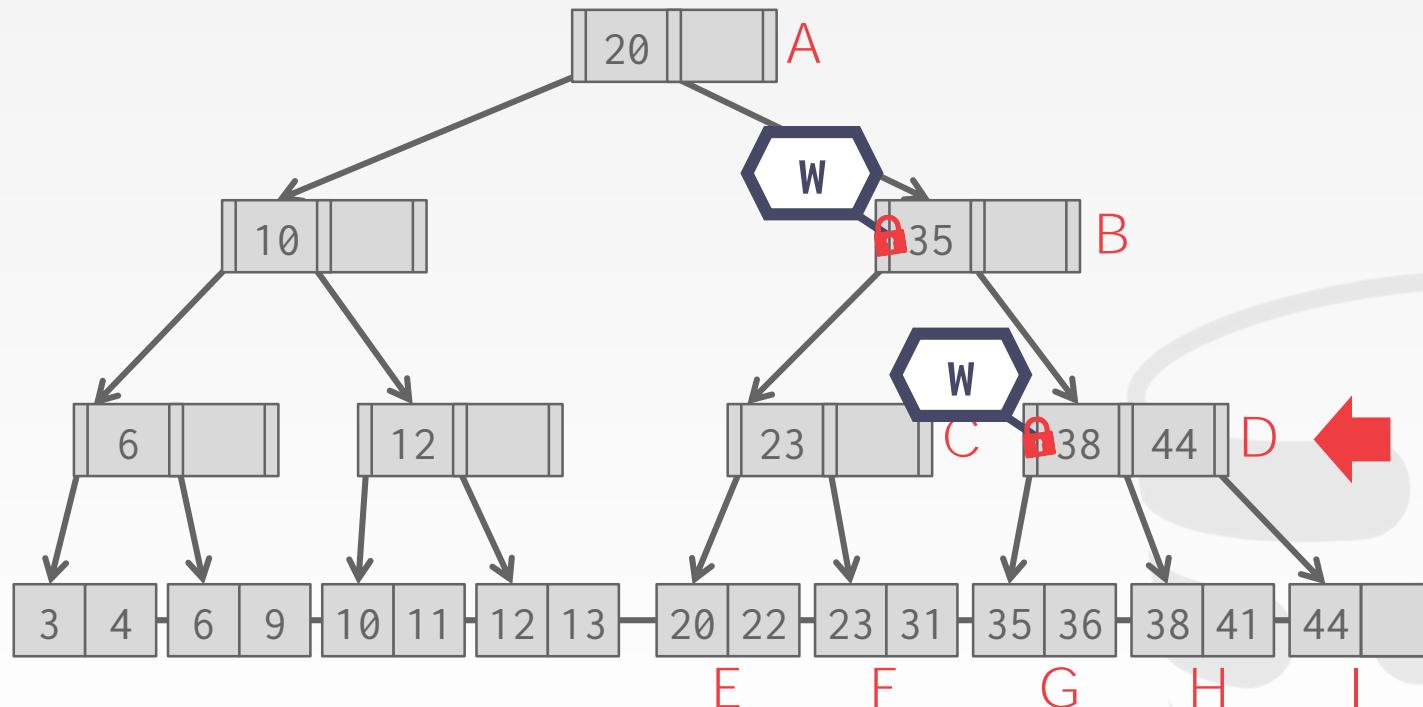
EXAMPLE #3 – INSERT 45



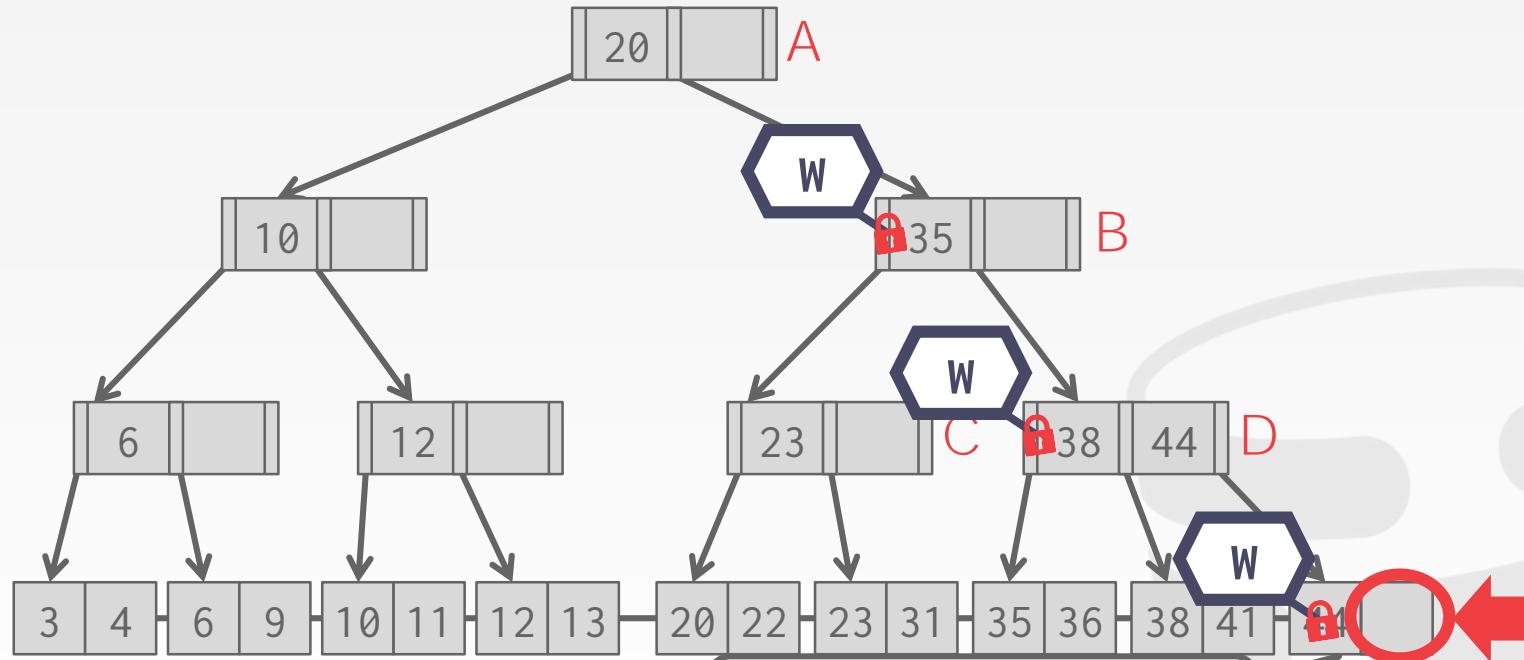
EXAMPLE #3 – INSERT 45



EXAMPLE #3 – INSERT 45

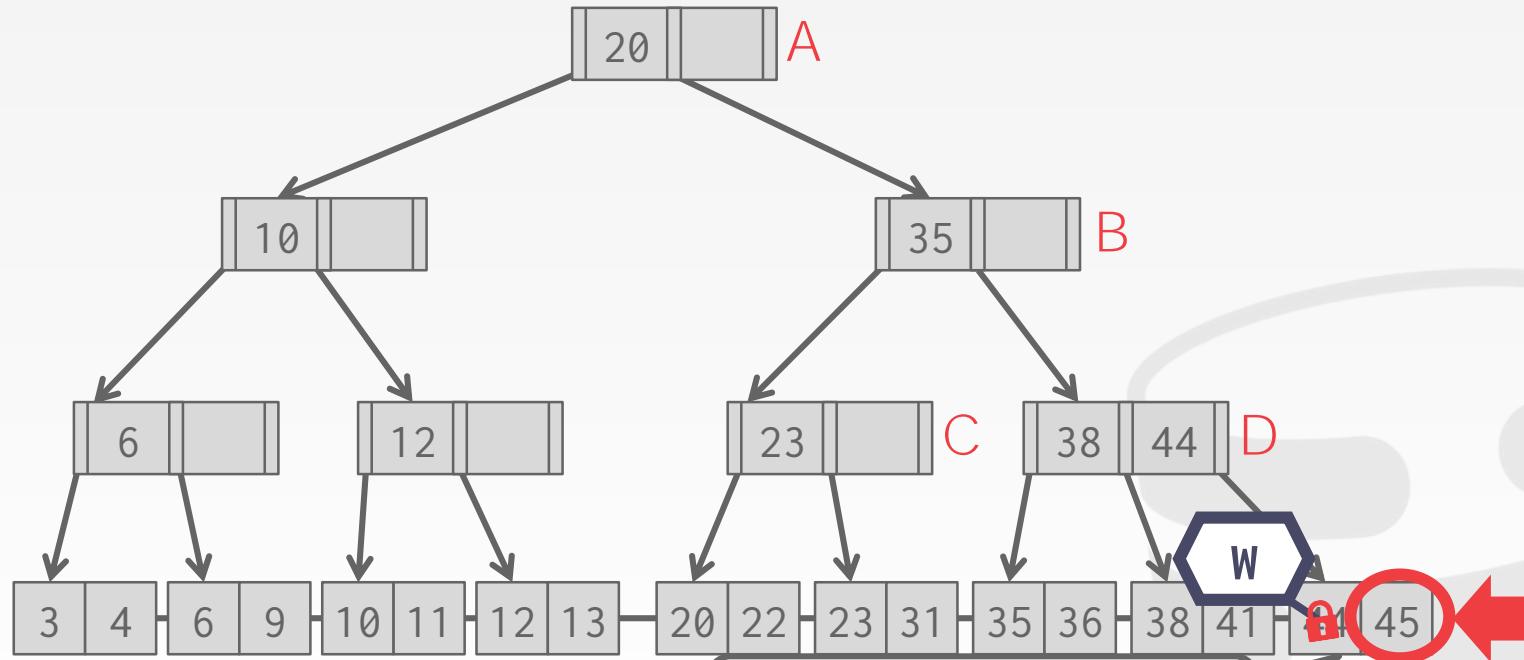


EXAMPLE #3 – INSERT 45



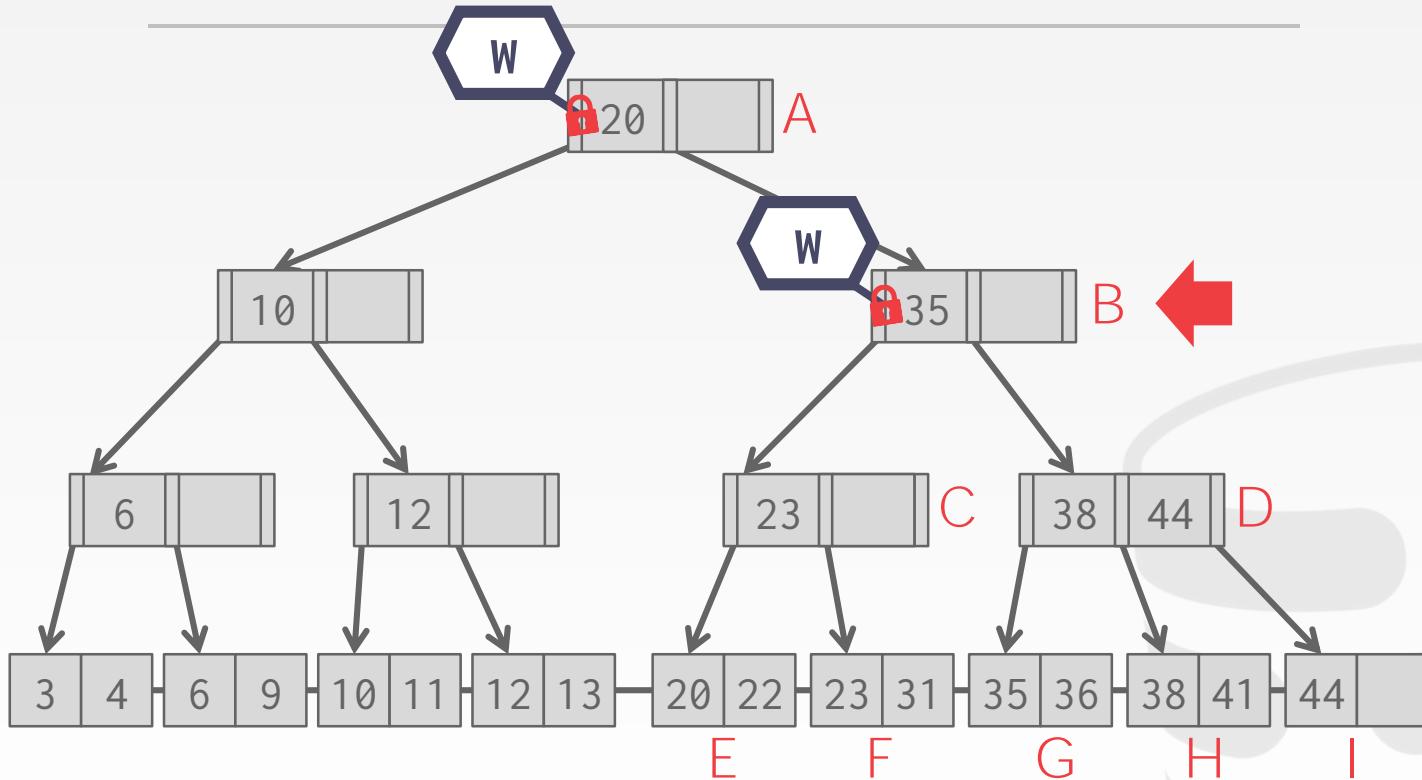
*Node I won't split, so we
can release B+D.*

EXAMPLE #3 – INSERT 45

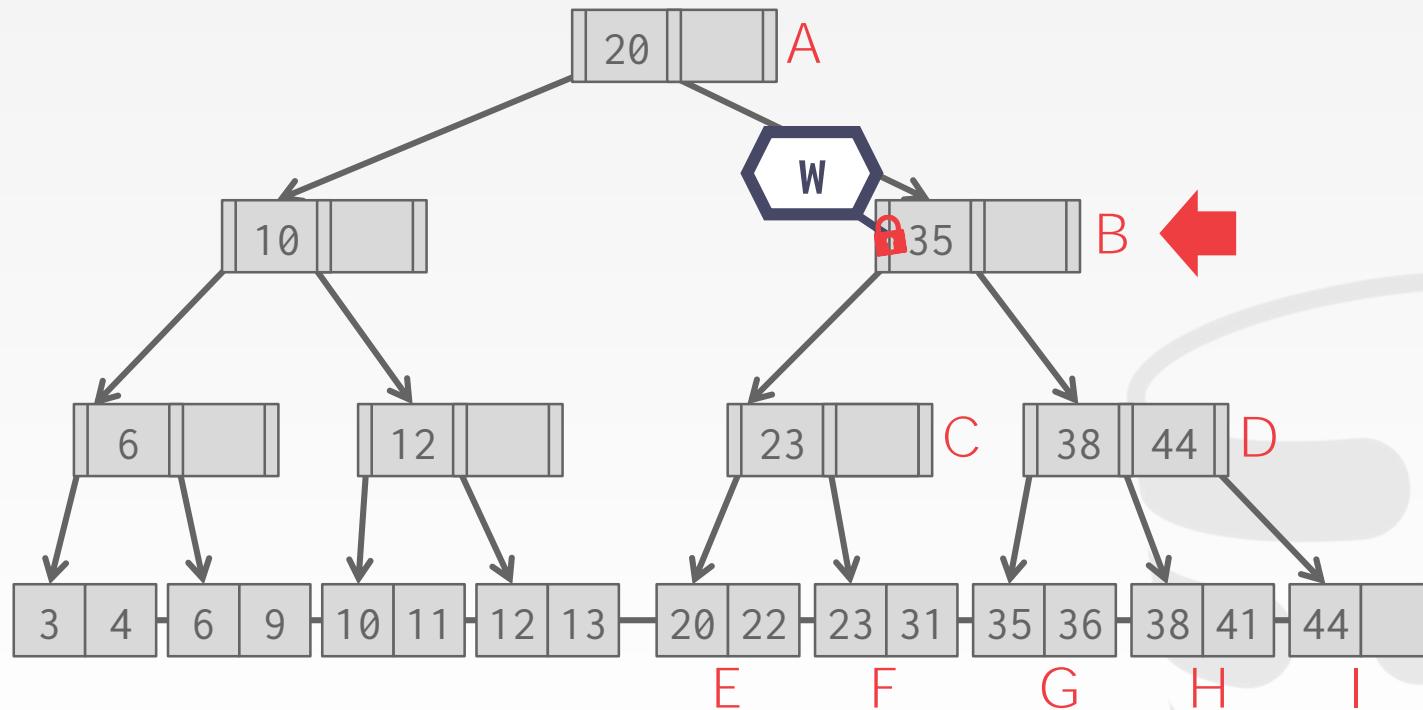


*Node I won't split, so we
can release B+D.*

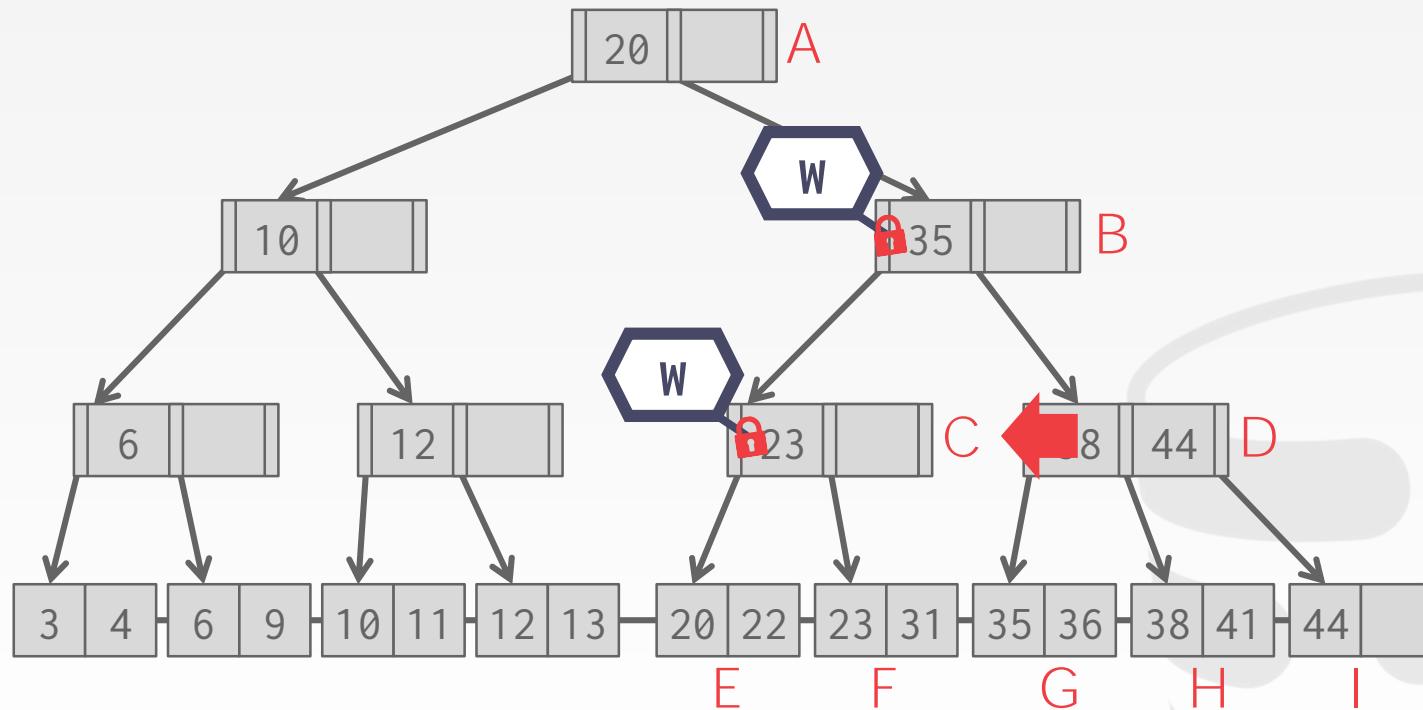
EXAMPLE #4 – INSERT 25



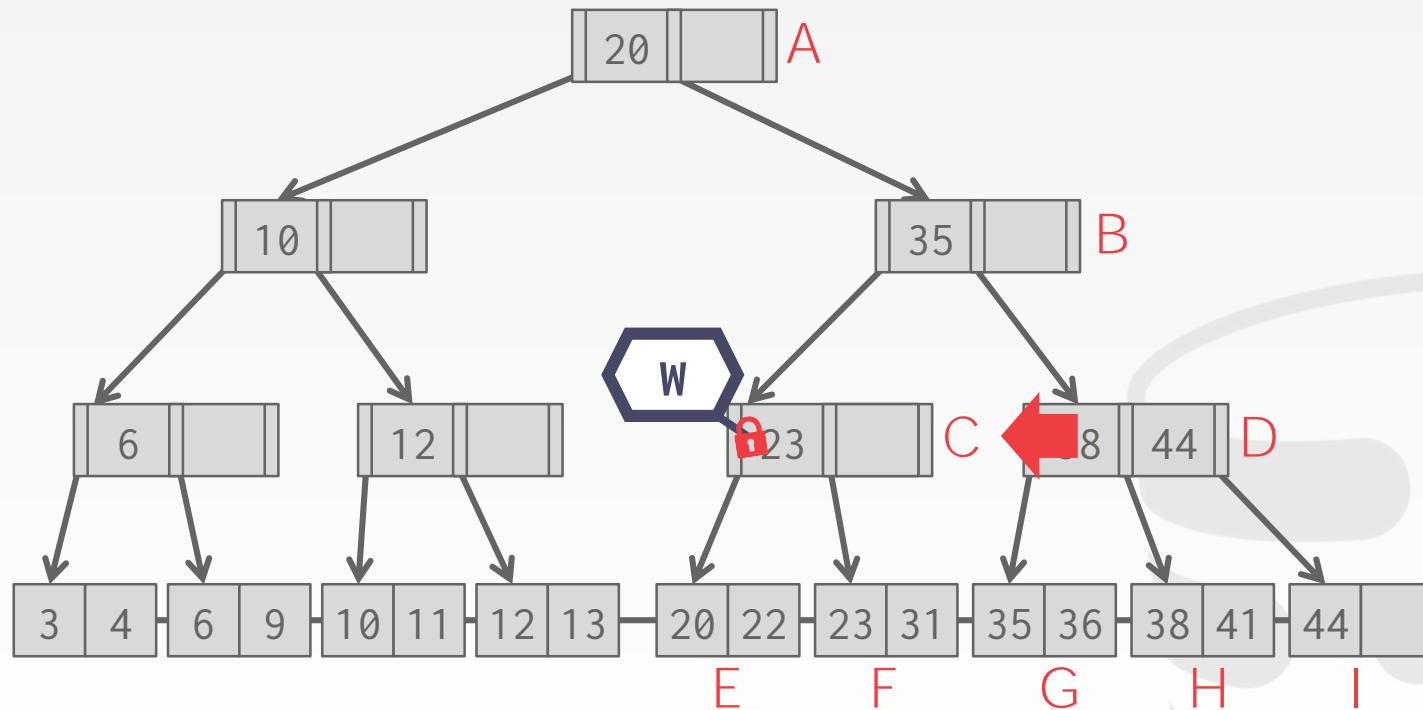
EXAMPLE #4 – INSERT 25



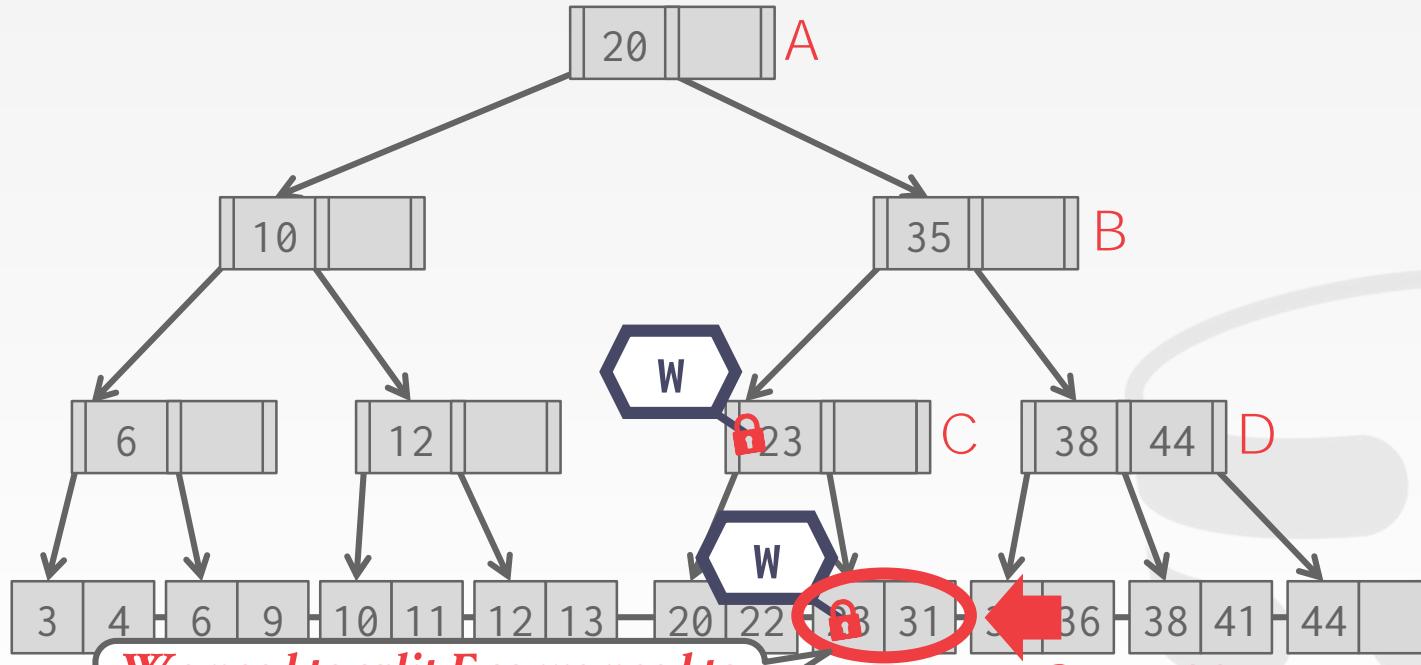
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

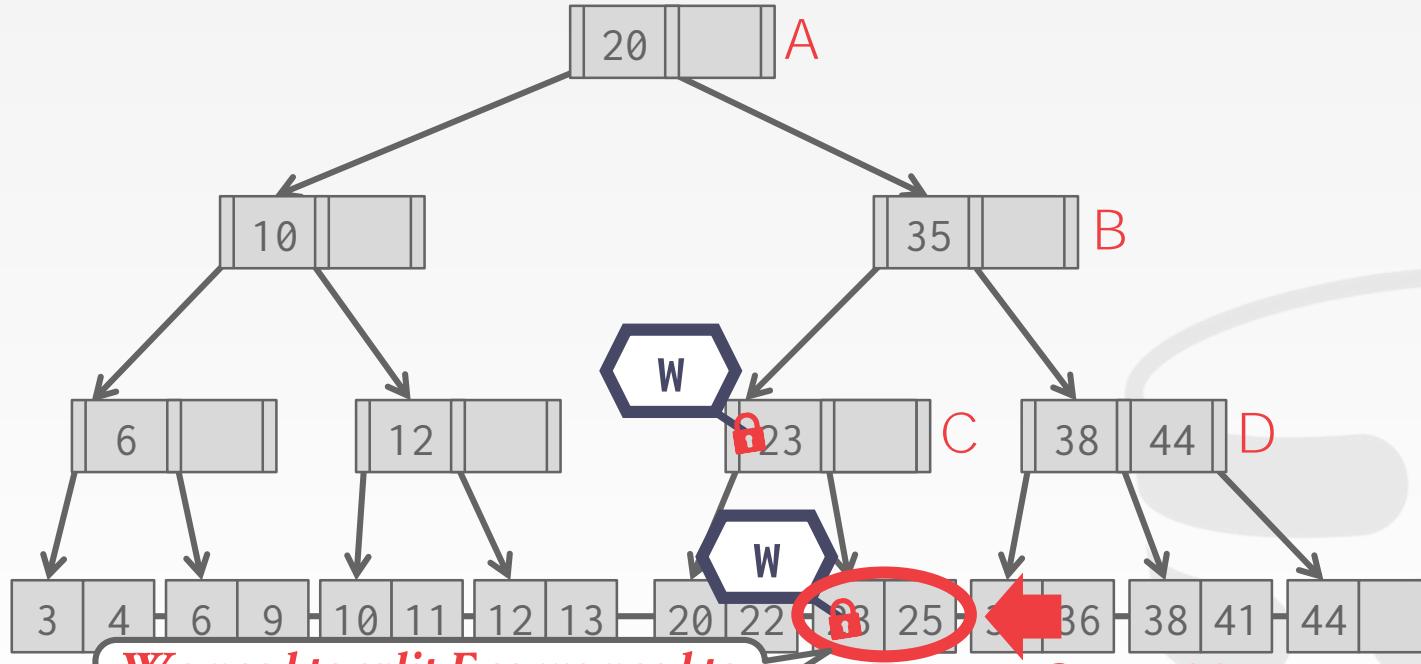


EXAMPLE #4 – INSERT 25

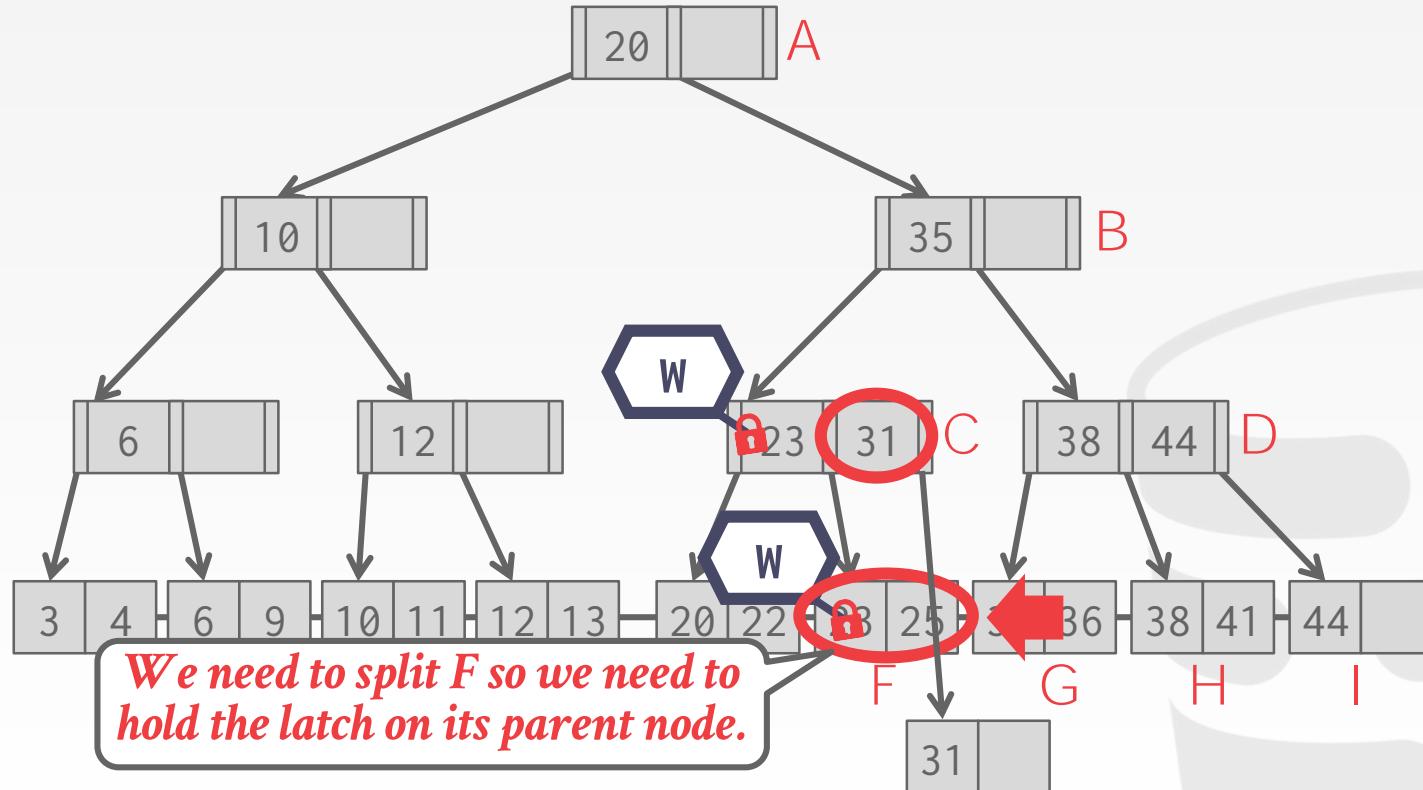


We need to split F so we need to hold the latch on its parent node.

EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



OBSERVATION

What was the first step that all the update examples did on the B+Tree?

Delete 38



Insert 45



Insert 25



OBSERVATION

What was the first step that all the update examples did on the B+Tree?

Taking a write latch on the root every time becomes a bottleneck with higher concurrency.

Can we do better?



BETTER LATCHING ALGORITHM

Assume that the leaf node is safe.

Use read latches and crabbing to reach it, and then verify that it is safe.

If leaf is not safe, then do previous algorithm using write latches.

Acta Informatica 9, 1–21 (1977)

 © by Springer-Verlag 1977

Concurrency of Operations on B-Trees

R. Bayer * and M. Schkolnick
IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on B-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock can be high.

Recently, some basic questions have been raised as to whether these structures can support concurrent operations. In this paper, we examine the problem of concurrent access to B-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

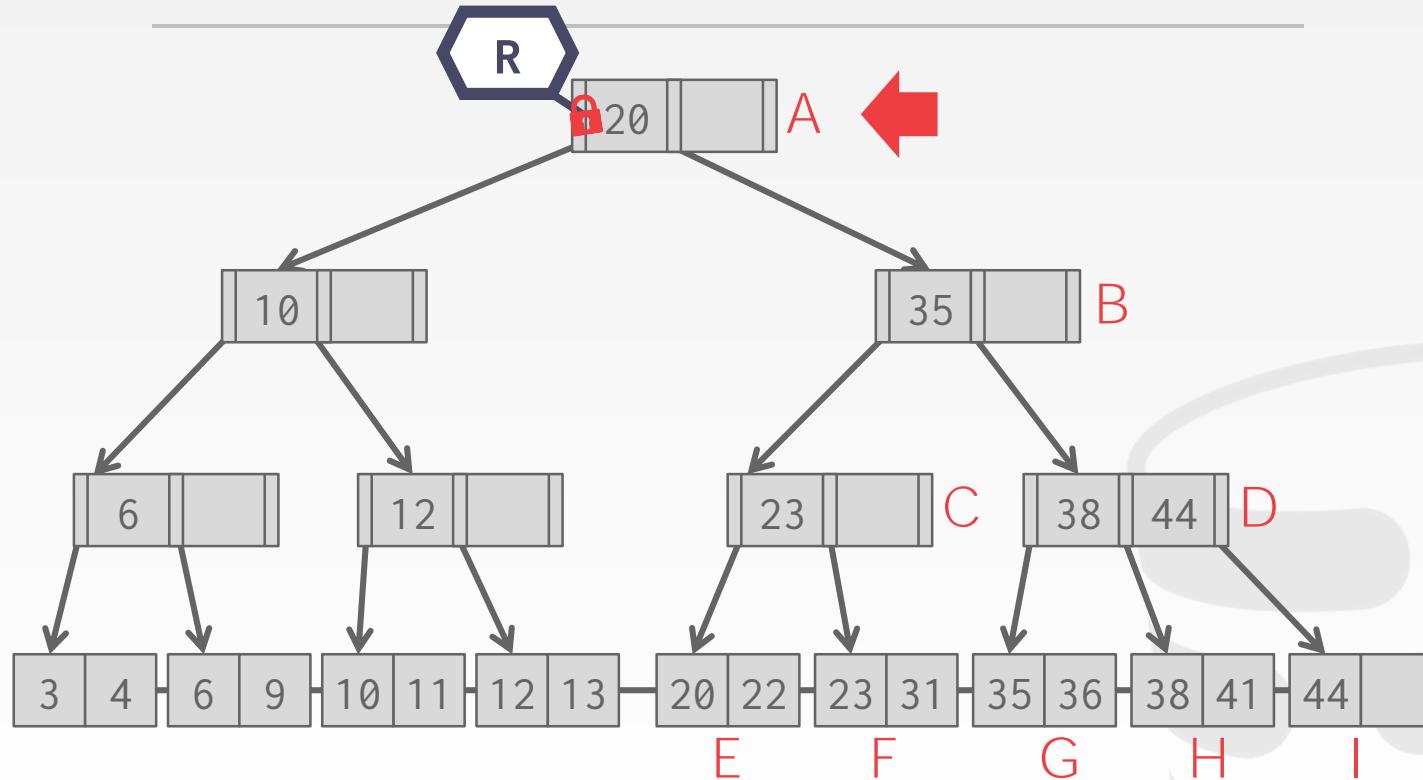
1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as B-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

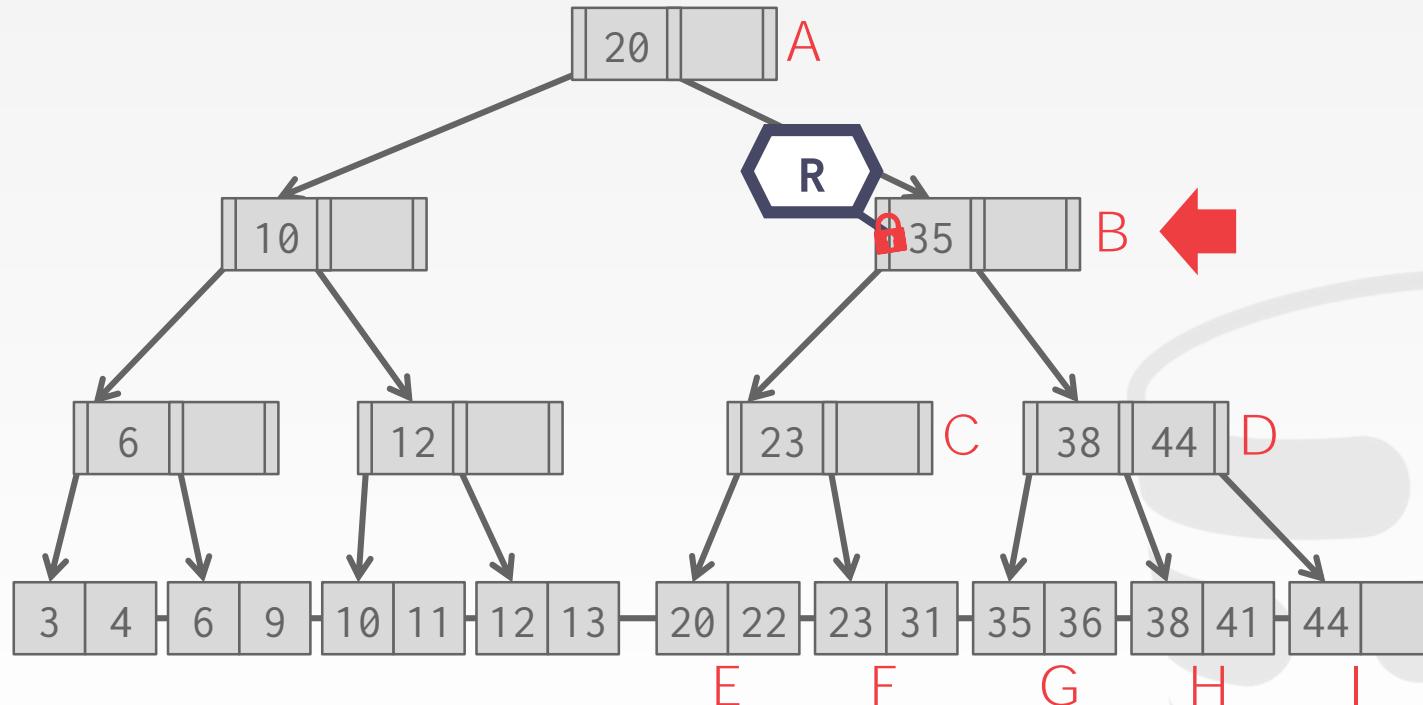
An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

* Permanent address: Institut für Informatik der Technischen Universität München, Arcistr. 21, D-8000 München 2, Germany (Fed. Rep.)

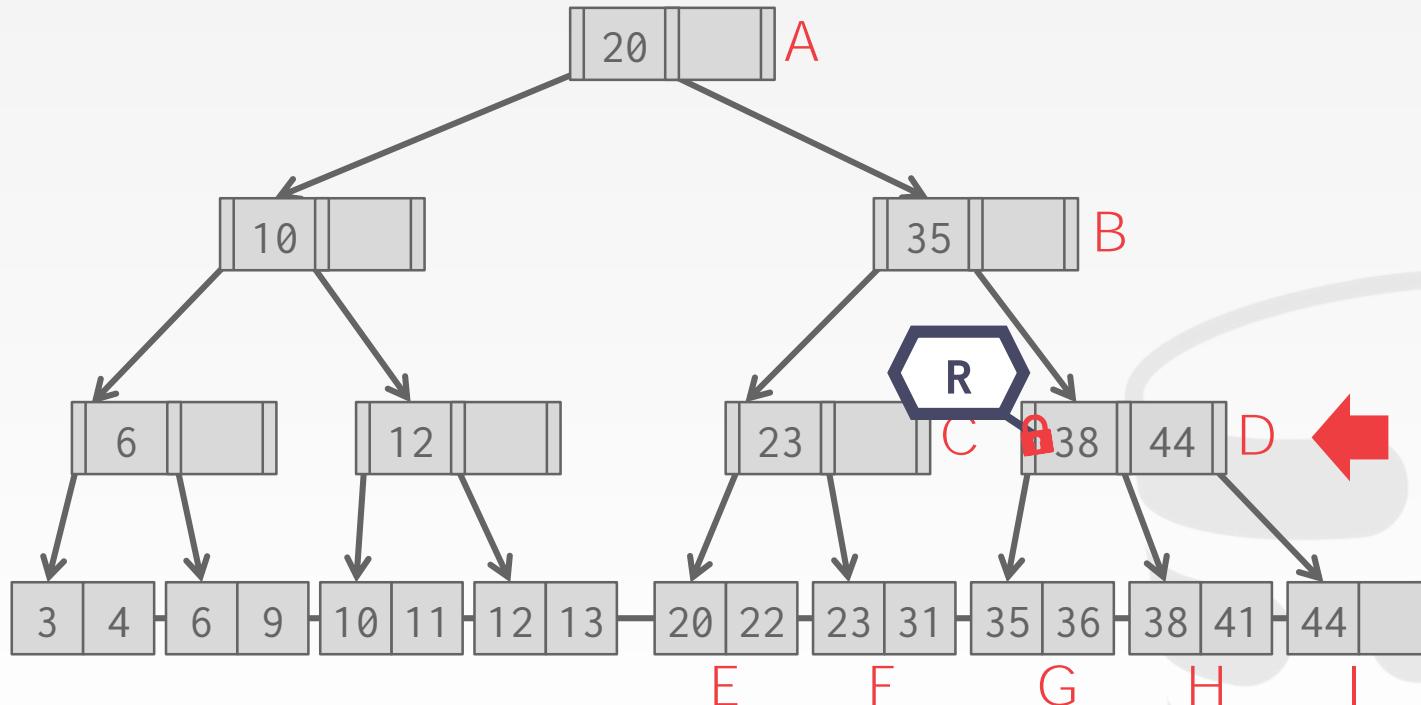
EXAMPLE #2 – DELETE 38



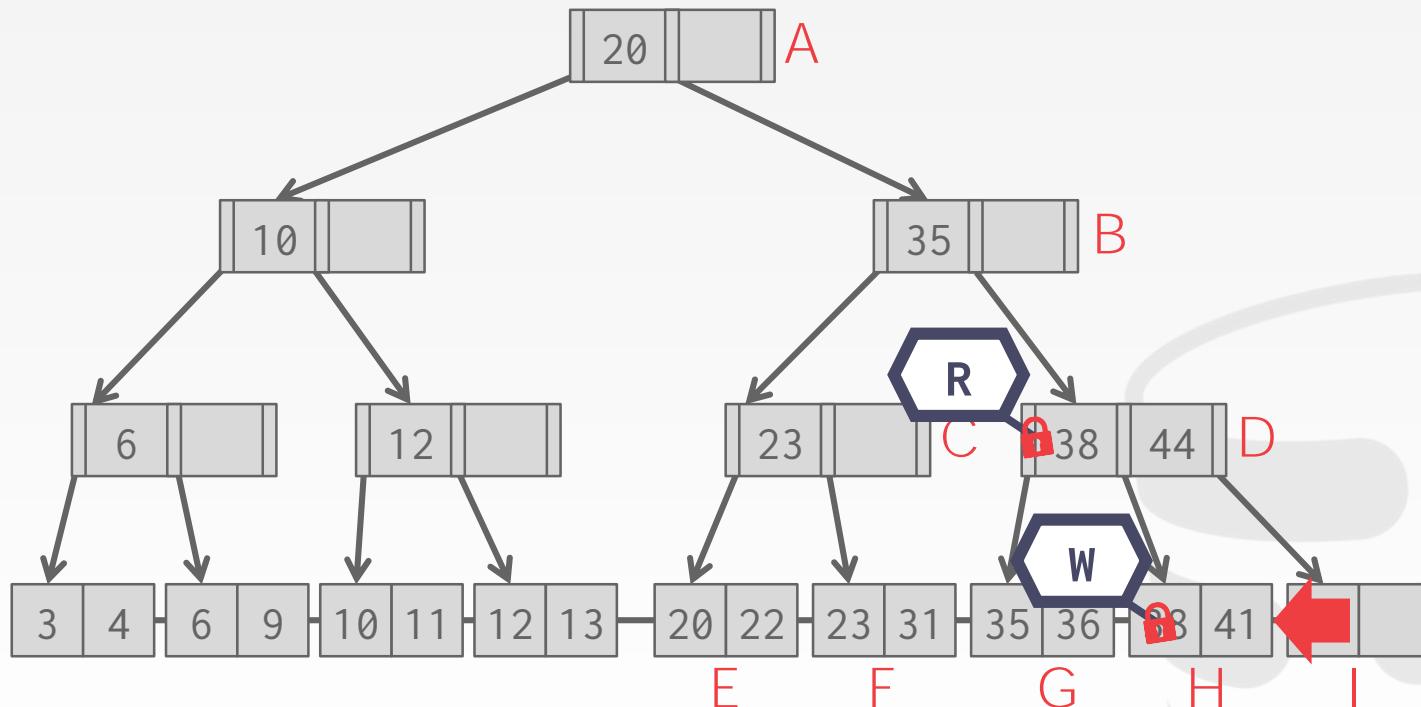
EXAMPLE #2 – DELETE 38



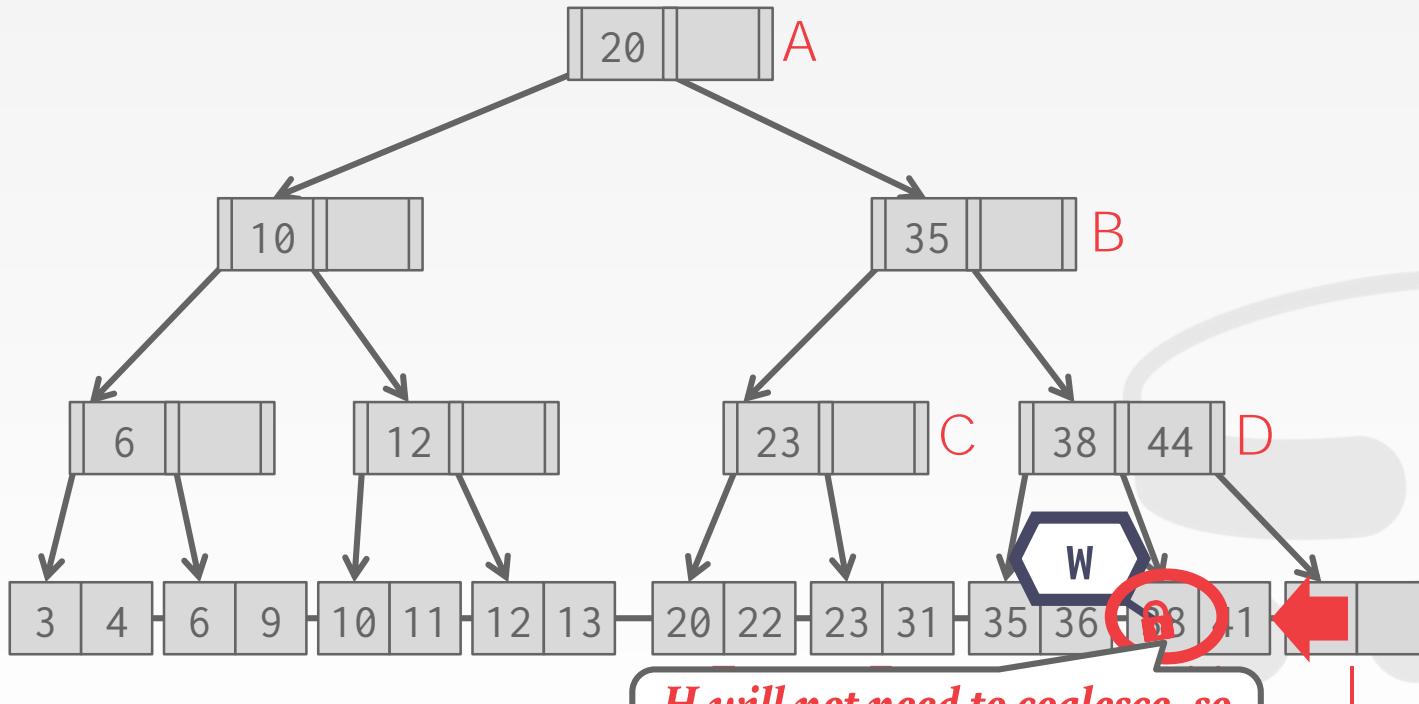
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38

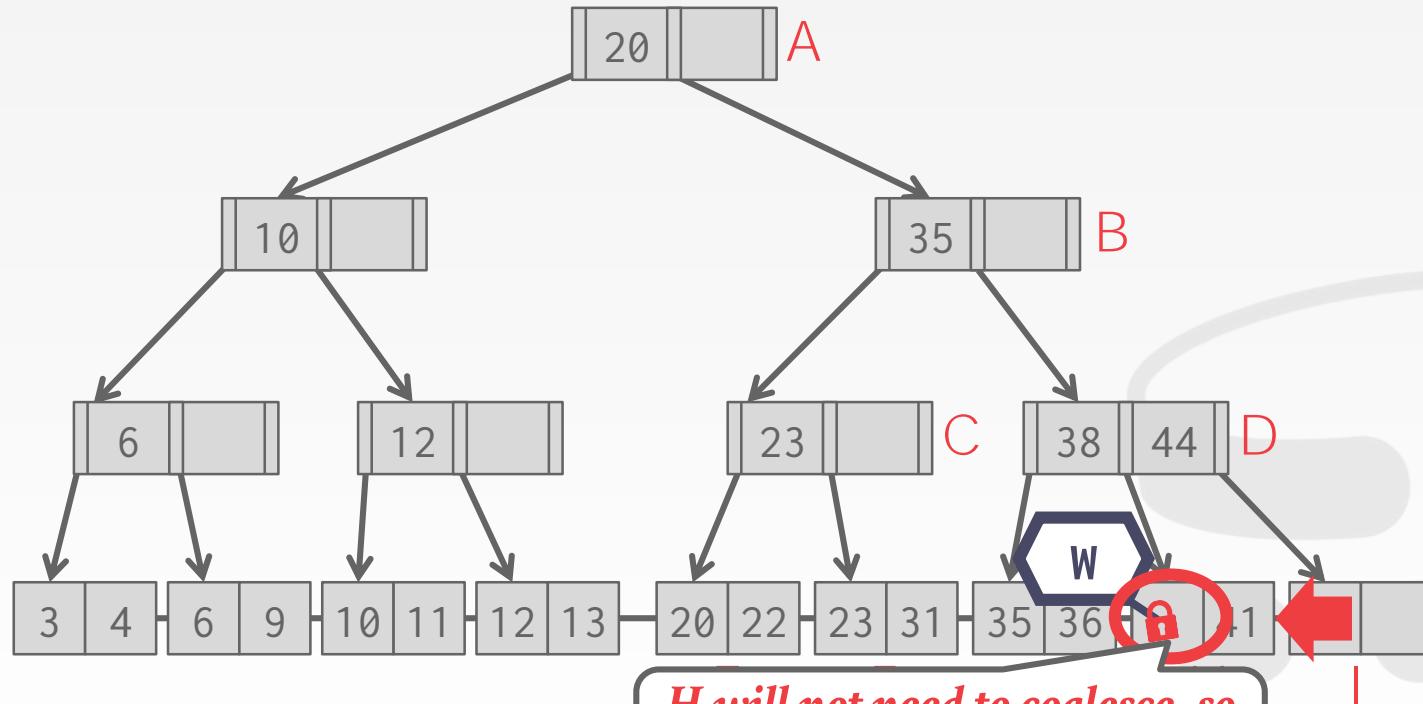


EXAMPLE #2 – DELETE 38



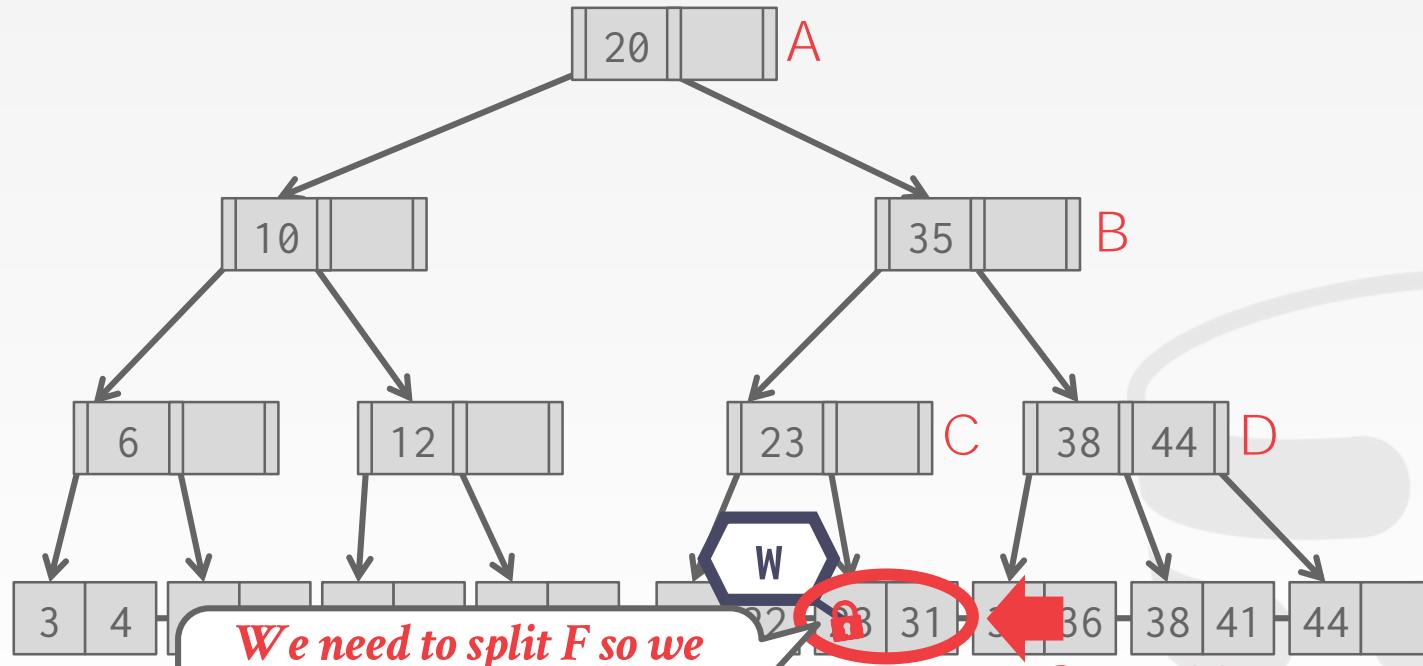
*H will not need to coalesce, so
we're safe!*

EXAMPLE #2 – DELETE 38



*H will not need to coalesce, so
we're safe!*

EXAMPLE #4 – INSERT 25



We need to split F so we
have to restart and re-
execute like before.

BETTER LATCHING ALGORITHM

Search: Same as before.

Insert/Delete:

- Set latches as if for search, get to leaf, and set **W** latch on leaf.
- If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

This approach optimistically assumes that only leaf node will be modified; if not, **R** latches set on the first pass to leaf are wasteful.

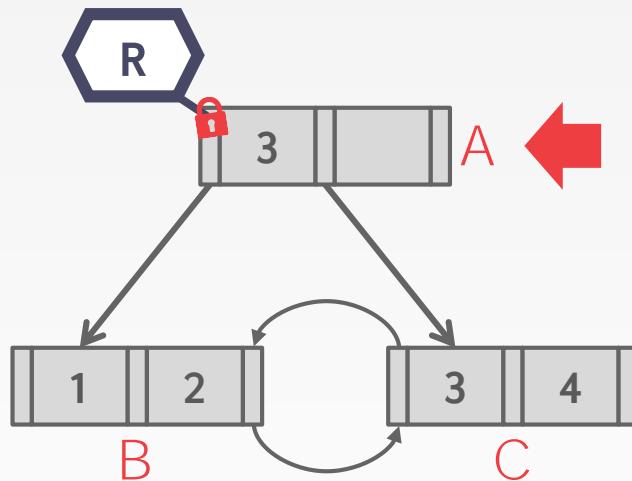
OBSERVATION

The threads in all the examples so far have acquired latches in a "top-down" manner.

- A thread can only acquire a latch from a node that is below its current node.
- If the desired latch is unavailable, the thread must wait until it becomes available.

But what if we want to move from one leaf node to another leaf node?

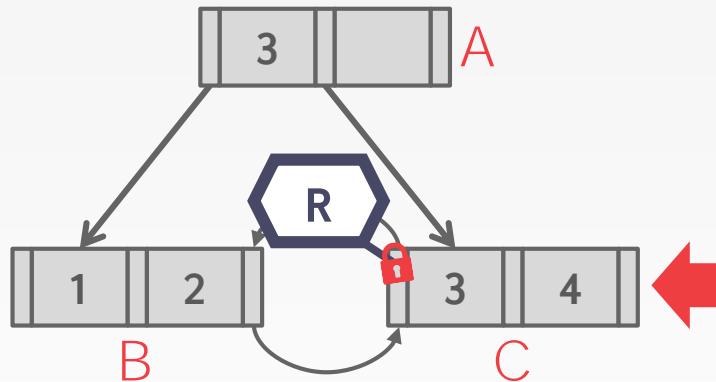
LEAF NODE SCAN EXAMPLE #1



T_1 : Find Keys < 4

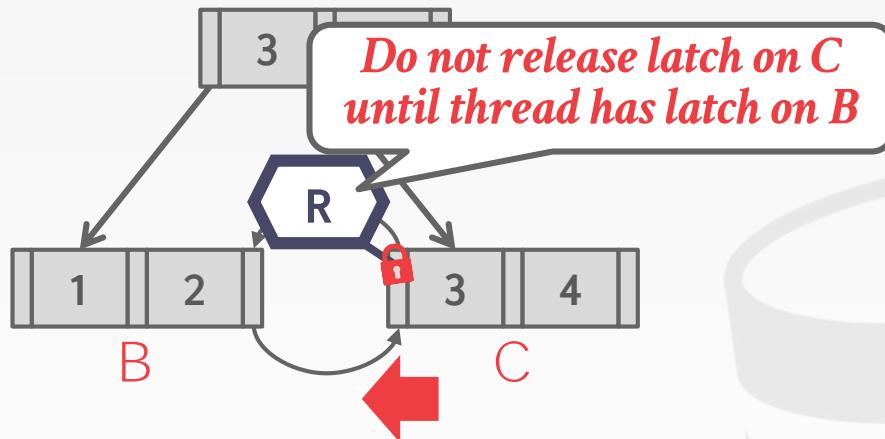
LEAF NODE SCAN EXAMPLE #1

T_1 : Find Keys < 4



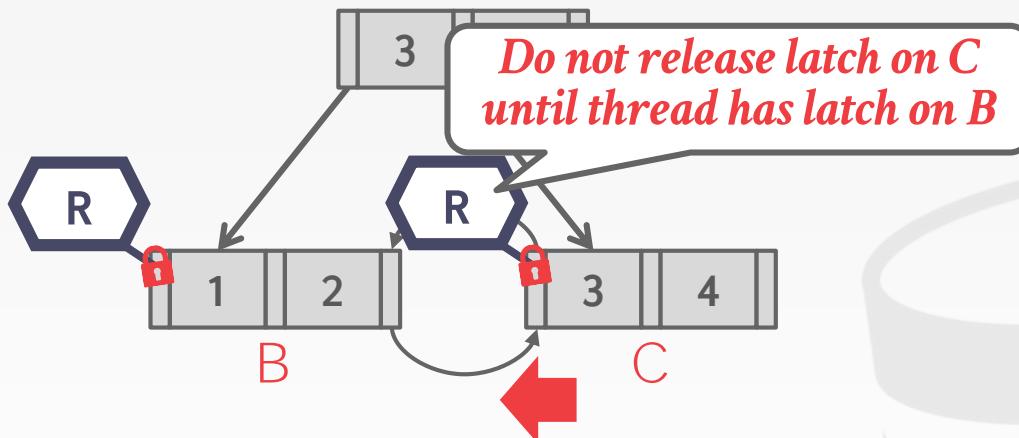
LEAF NODE SCAN EXAMPLE #1

T_1 : Find Keys < 4



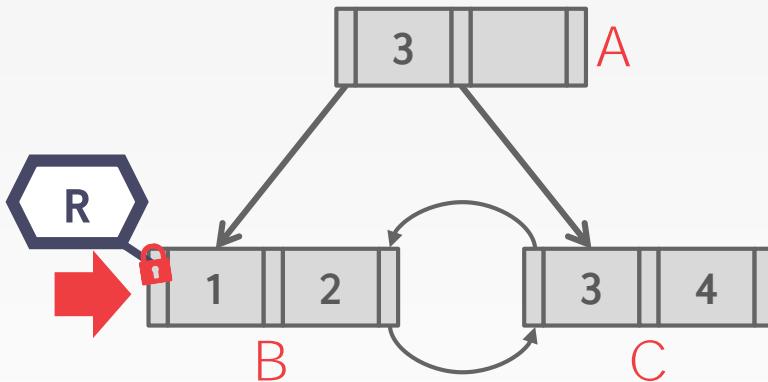
LEAF NODE SCAN EXAMPLE #1

T_1 : Find Keys < 4



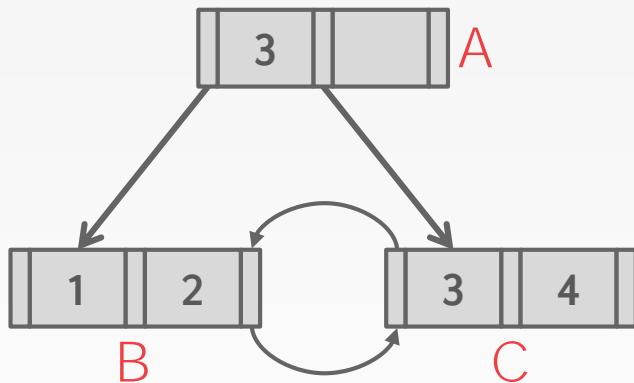
LEAF NODE SCAN EXAMPLE #1

T_1 : Find Keys < 4

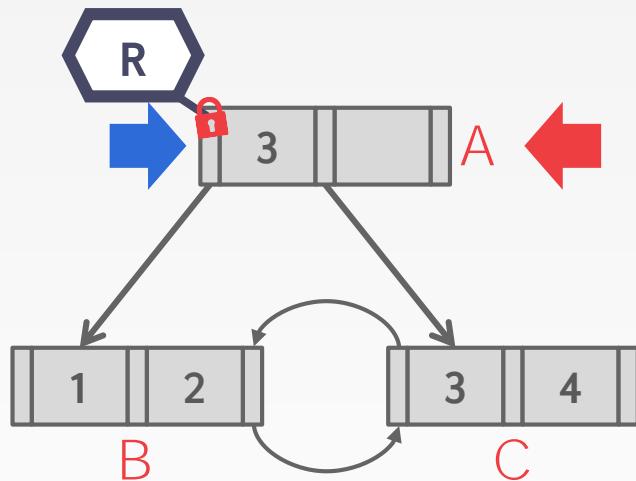


LEAF NODE SCAN EXAMPLE #2

T_1 : Find Keys < 4
 T_2 : Find Keys > 1

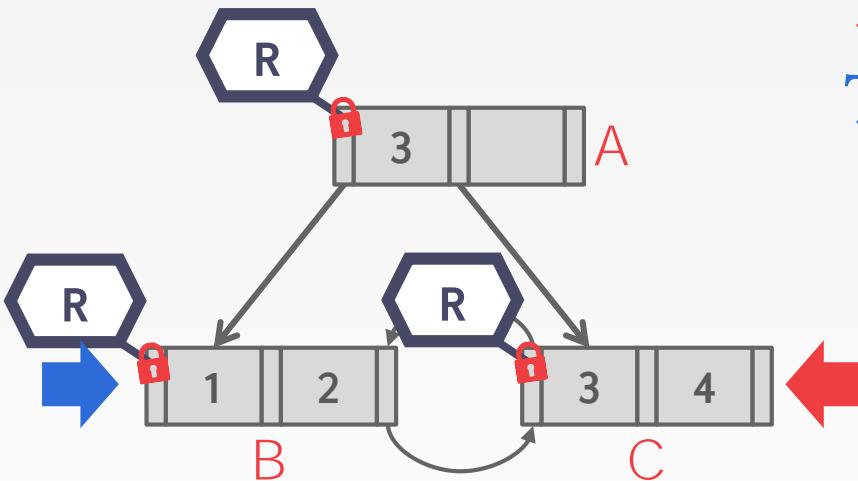


LEAF NODE SCAN EXAMPLE #2



T_1 : Find Keys < 4
 T_2 : Find Keys > 1

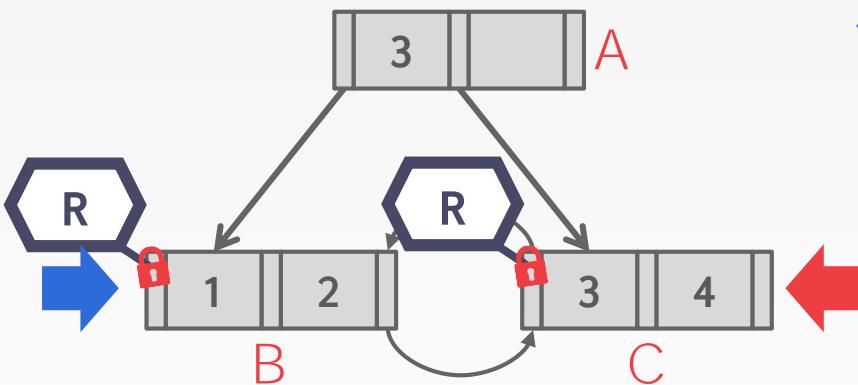
LEAF NODE SCAN EXAMPLE #2



T_1 : Find Keys < 4
 T_2 : Find Keys > 1

LEAF NODE SCAN EXAMPLE #2

T_1 : Find Keys < 4
 T_2 : Find Keys > 1

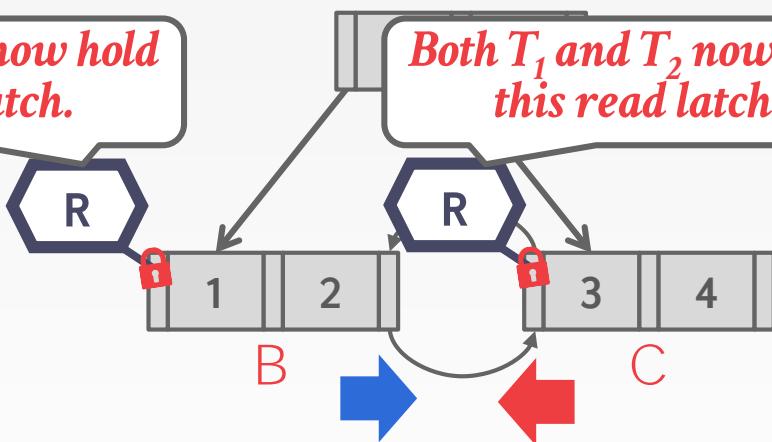


LEAF NODE SCAN EXAMPLE #2

T_1 : Find Keys < 4
 T_2 : Find Keys > 1

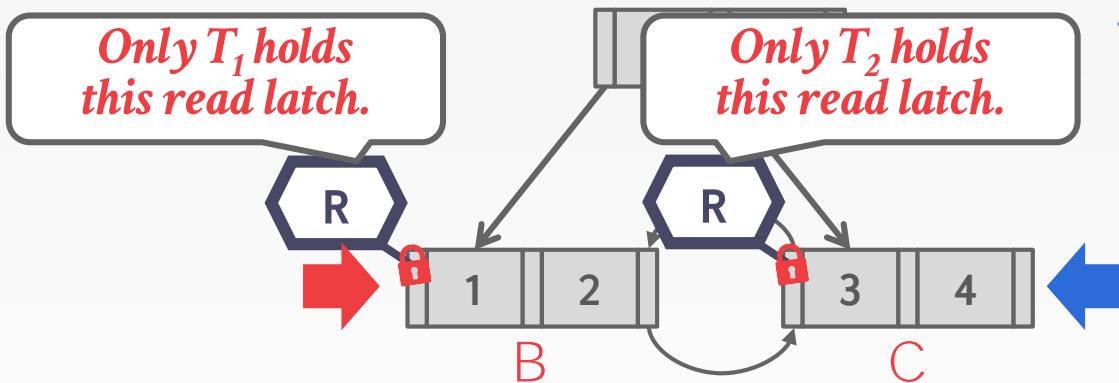
Both T_1 and T_2 now hold this read latch.

Both T_1 and T_2 now hold this read latch.

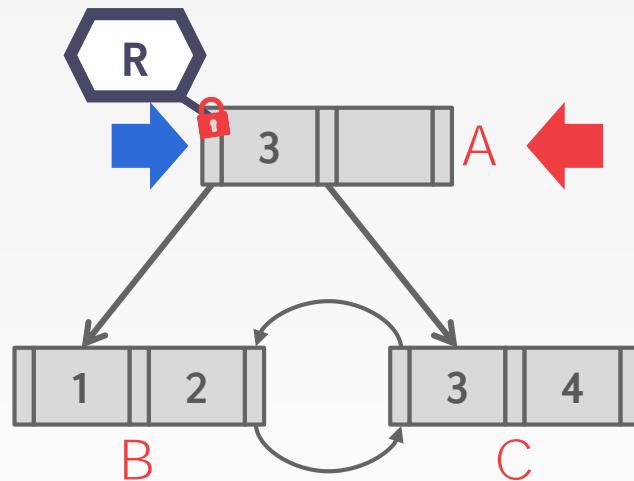


LEAF NODE SCAN EXAMPLE #2

T_1 : Find Keys < 4
 T_2 : Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

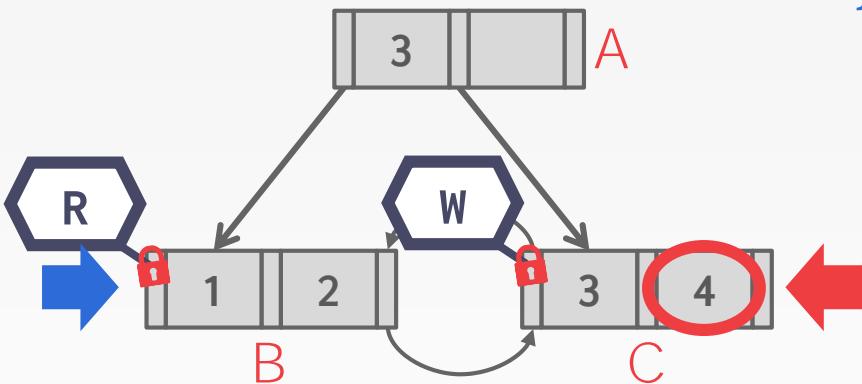


T_1 : Delete 4

T_2 : Find Keys > 1

LEAF NODE SCAN EXAMPLE #3

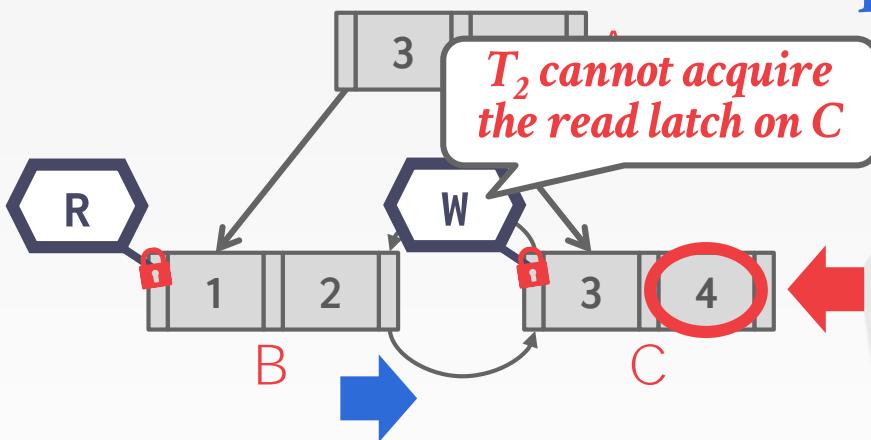
T_1 : Delete 4
 T_2 : Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1 : Delete 4

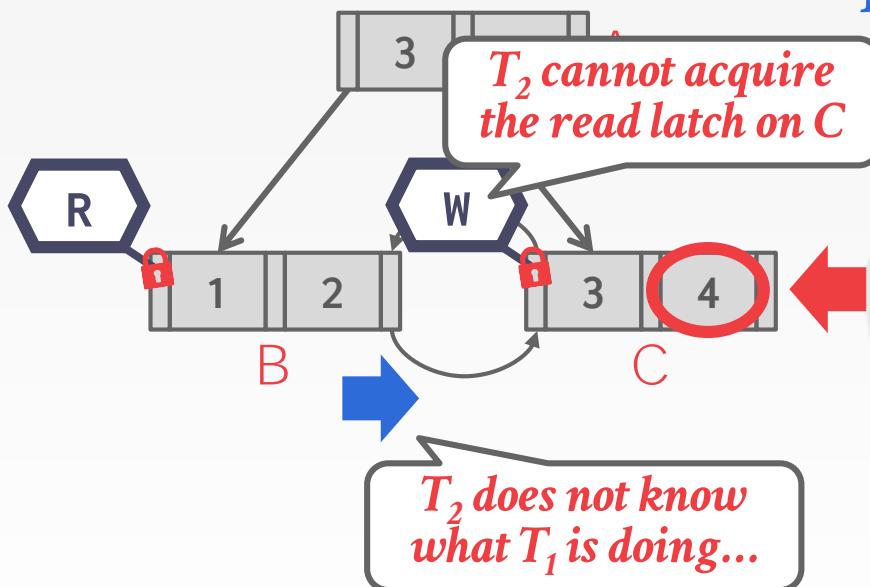
T_2 : Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1 : Delete 4

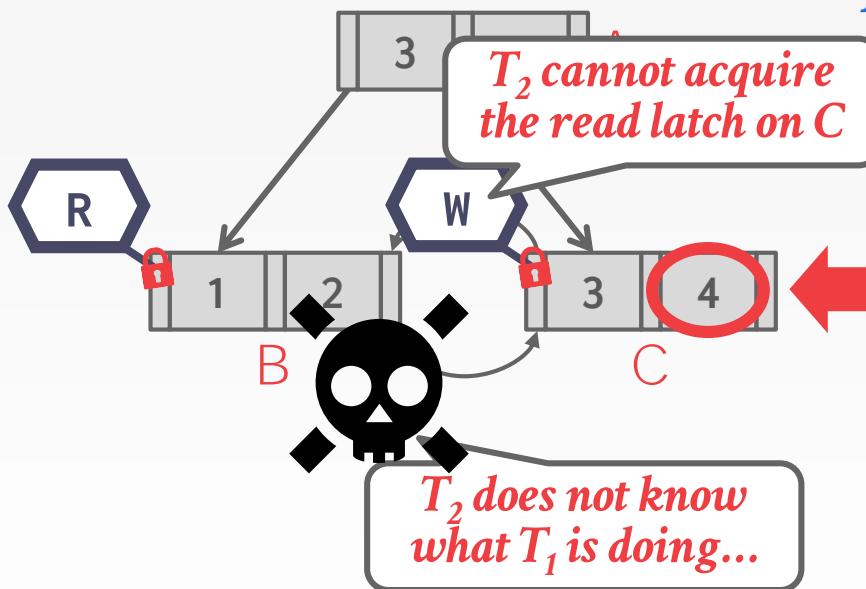
T_2 : Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1 : Delete 4

T_2 : Find Keys > 1

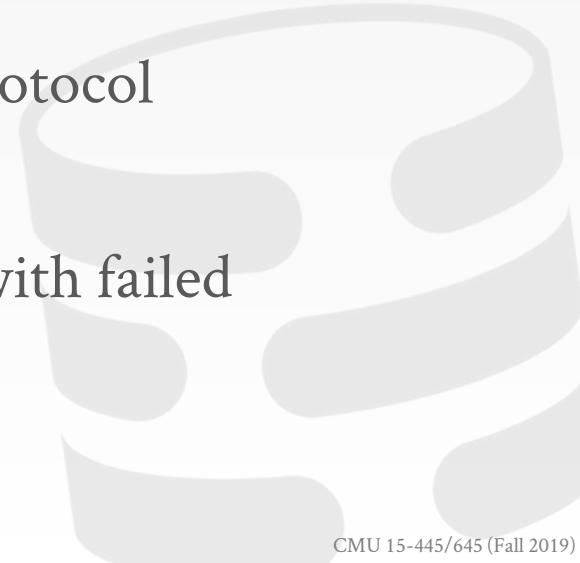


LEAF NODE SCANS

Latches do not support deadlock detection or avoidance. The only way we can deal with this problem is through coding discipline.

The leaf node sibling latch acquisition protocol must support a "no-wait" mode.

The DBMS's data structures must cope with failed latch acquisitions.

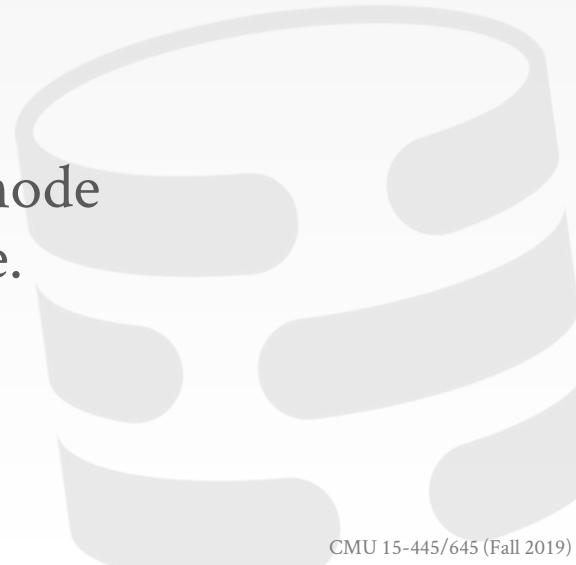


DELAYED PARENT UPDATES

Every time a leaf node overflows, we must update at least three nodes.

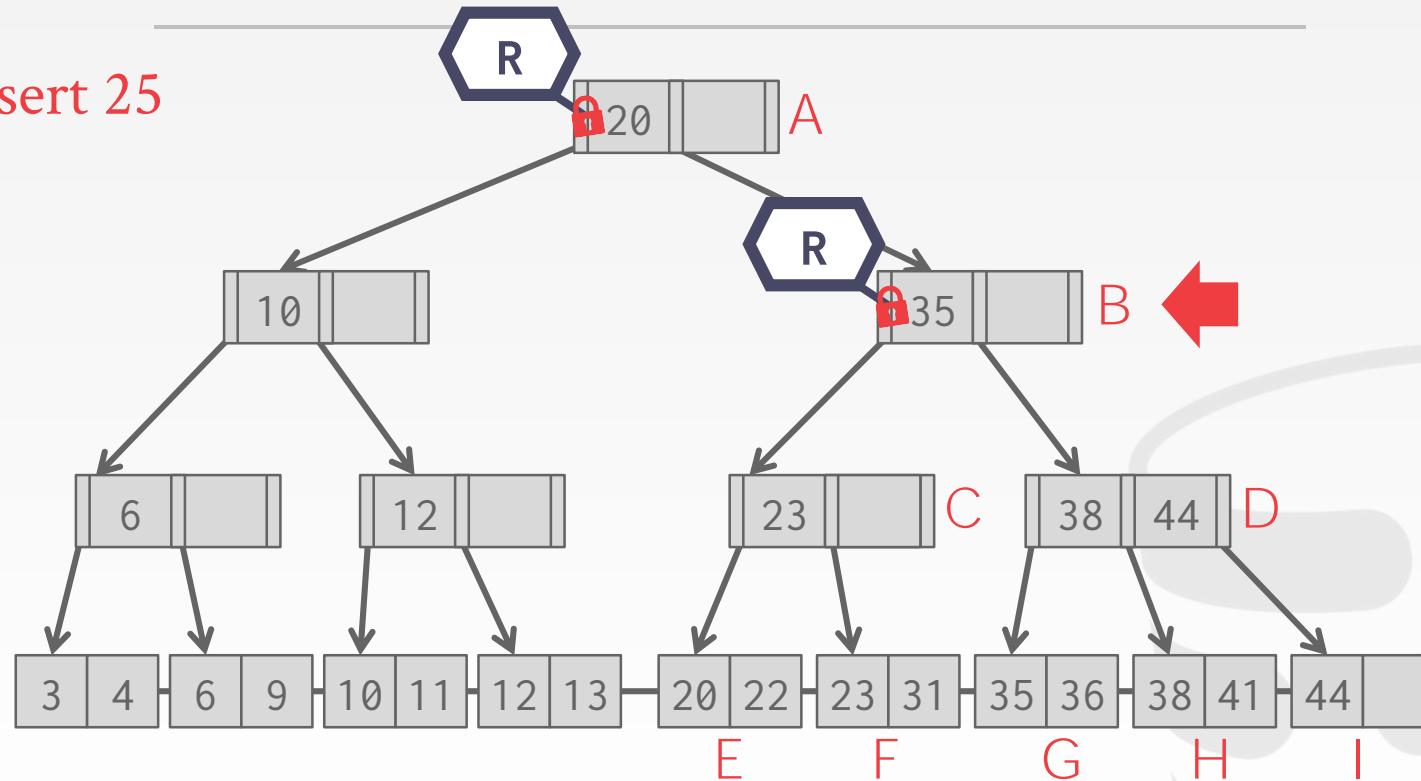
- The leaf node being split.
- The new leaf node being created.
- The parent node.

B^{link}-Tree Optimization: When a leaf node overflows, delay updating its parent node.



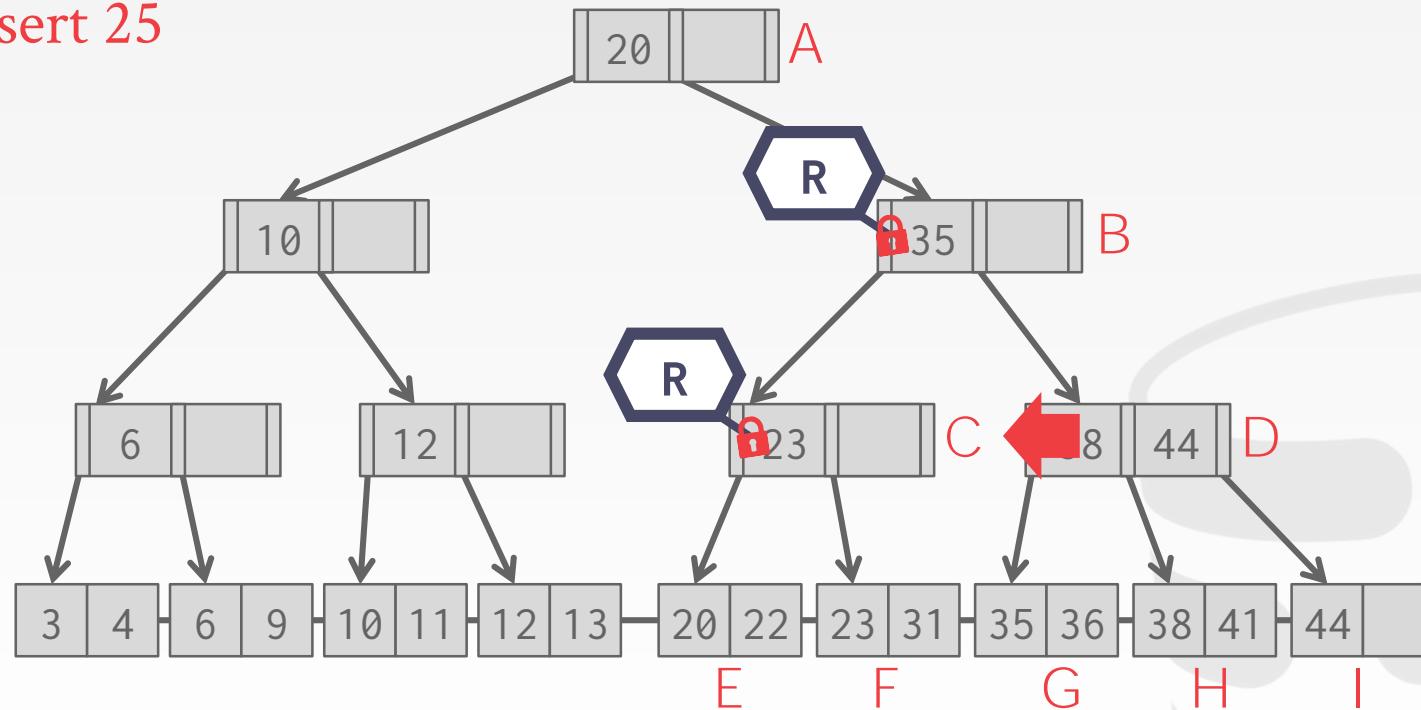
EXAMPLE #4 – INSERT 25

T₁: Insert 25



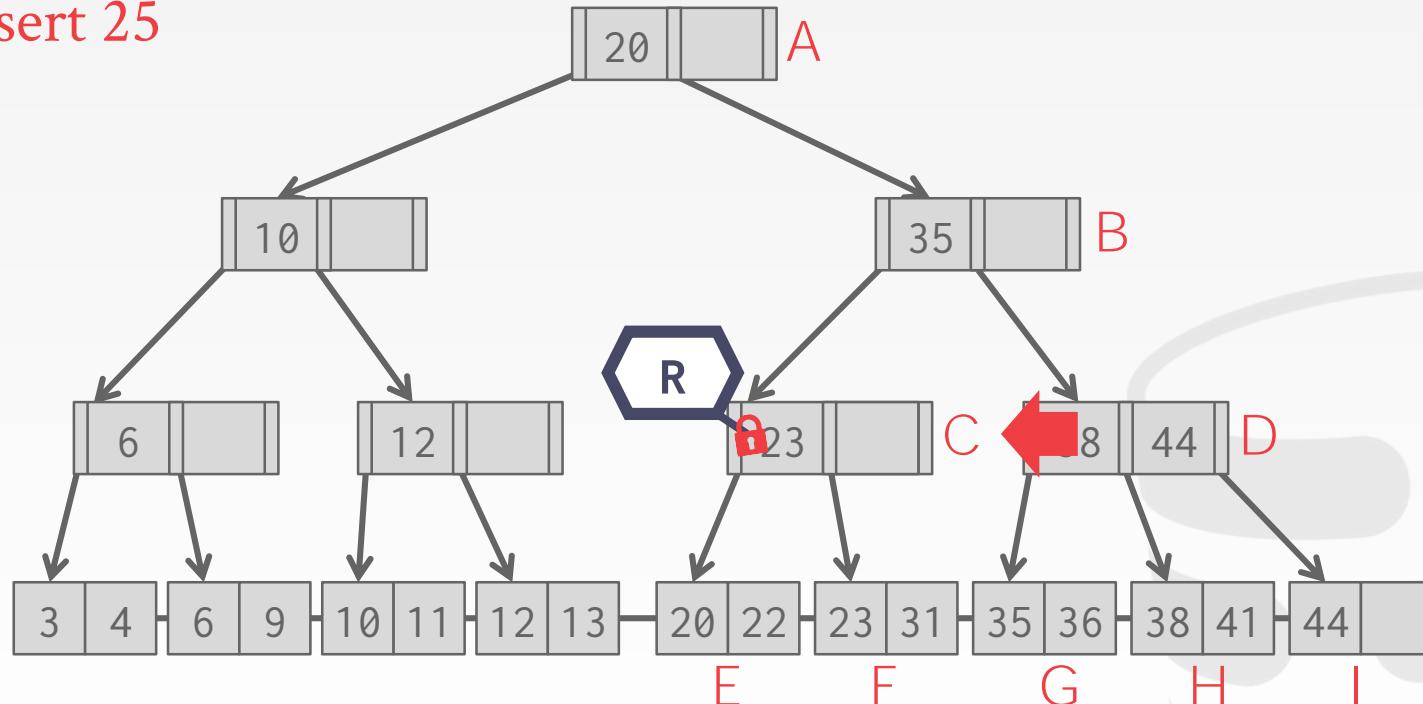
EXAMPLE #4 – INSERT 25

T₁: Insert 25



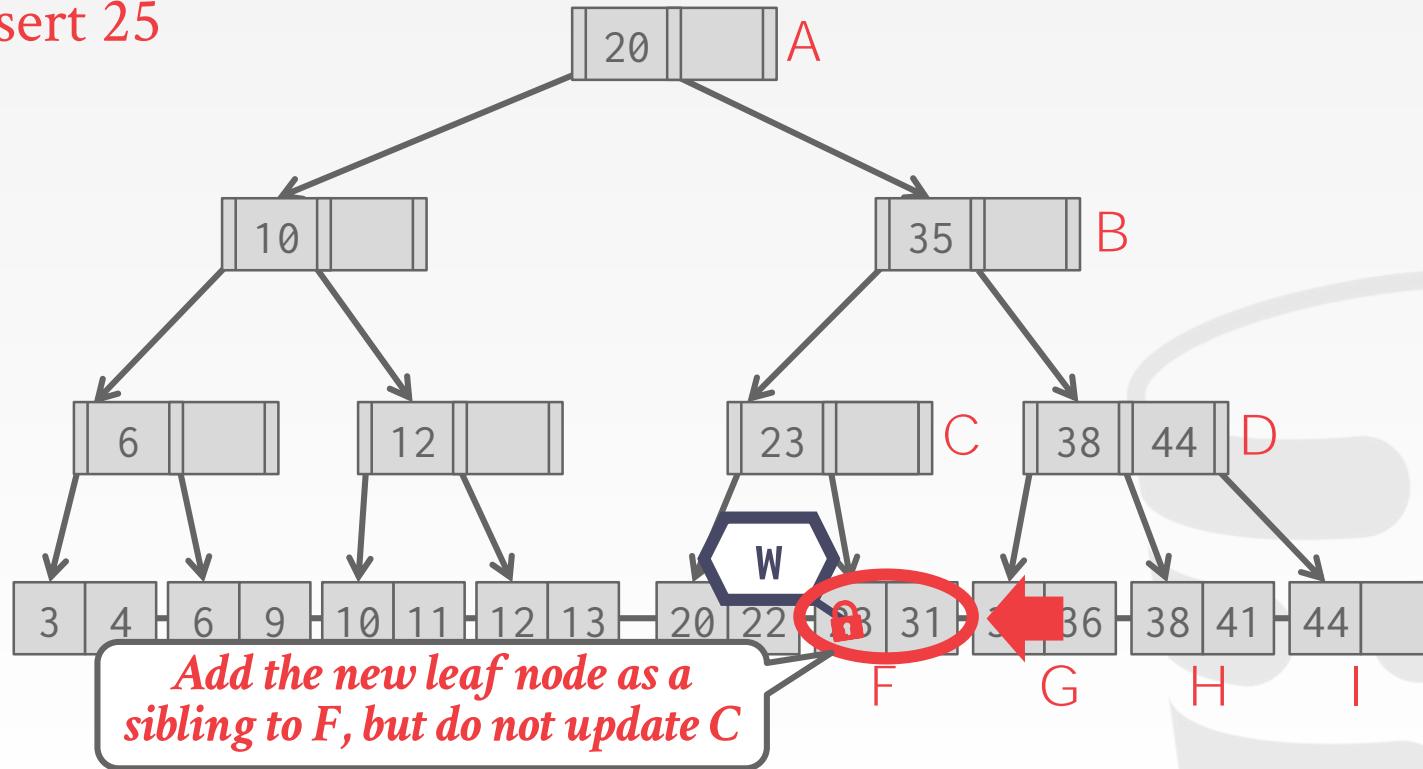
EXAMPLE #4 – INSERT 25

T₁: Insert 25



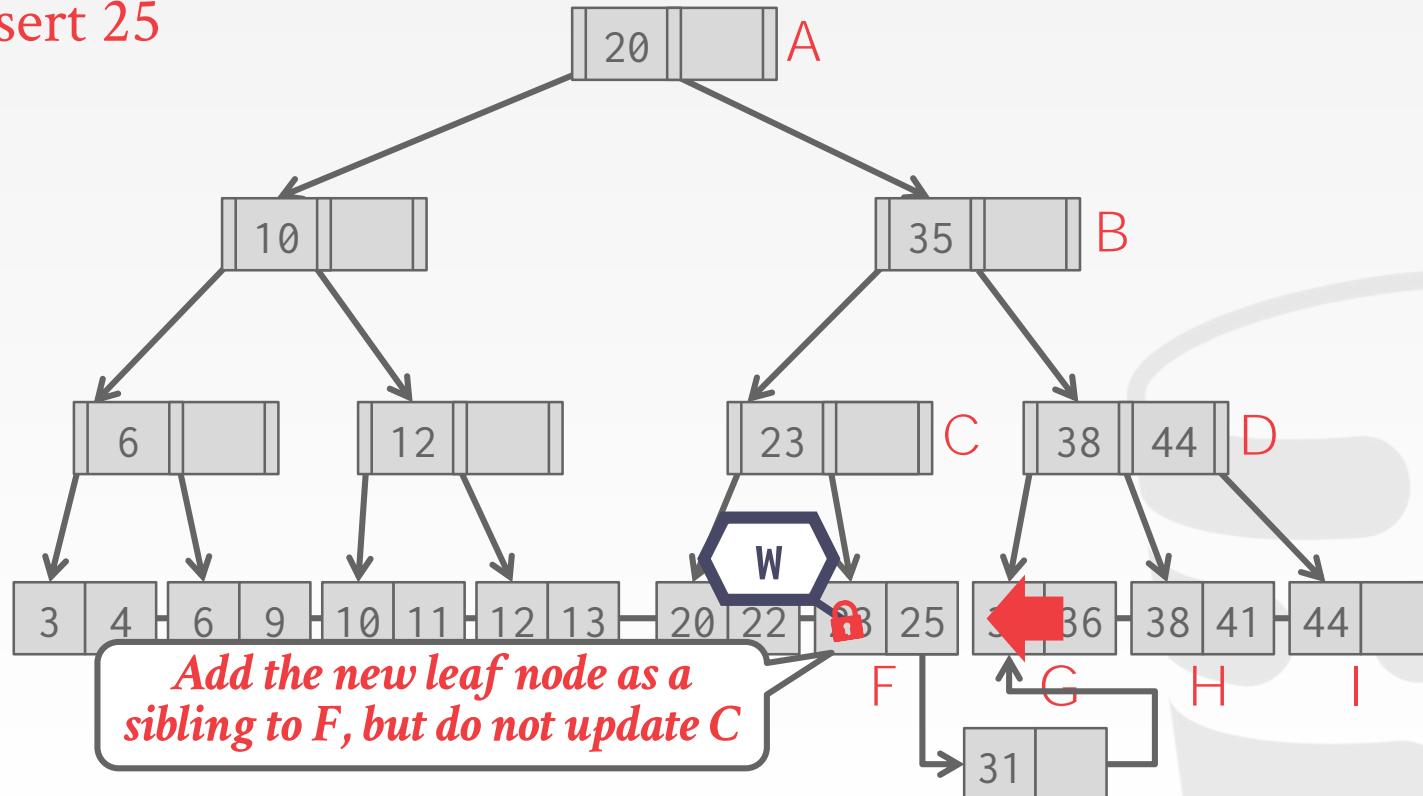
EXAMPLE #4 – INSERT 25

T₁: Insert 25



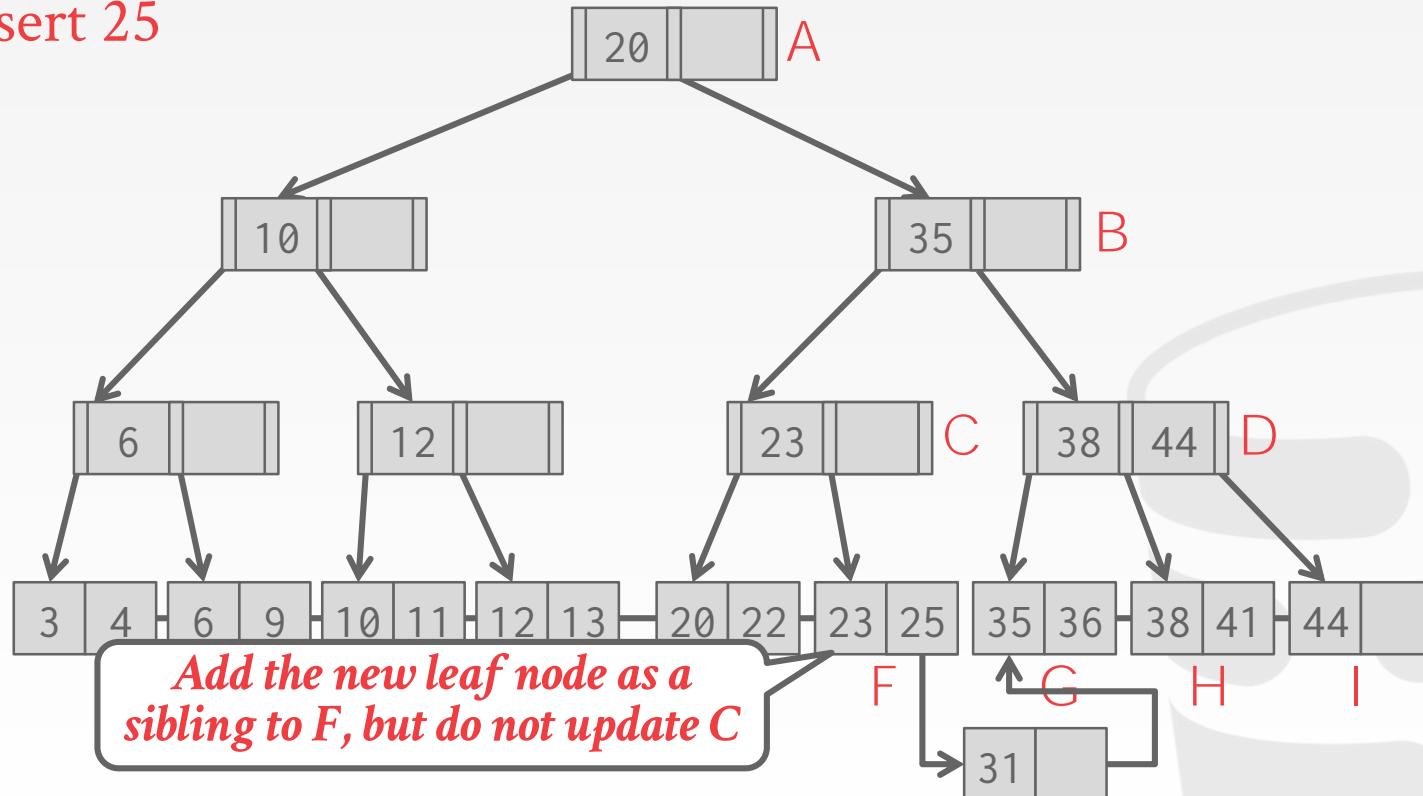
EXAMPLE #4 – INSERT 25

T_1 : Insert 25



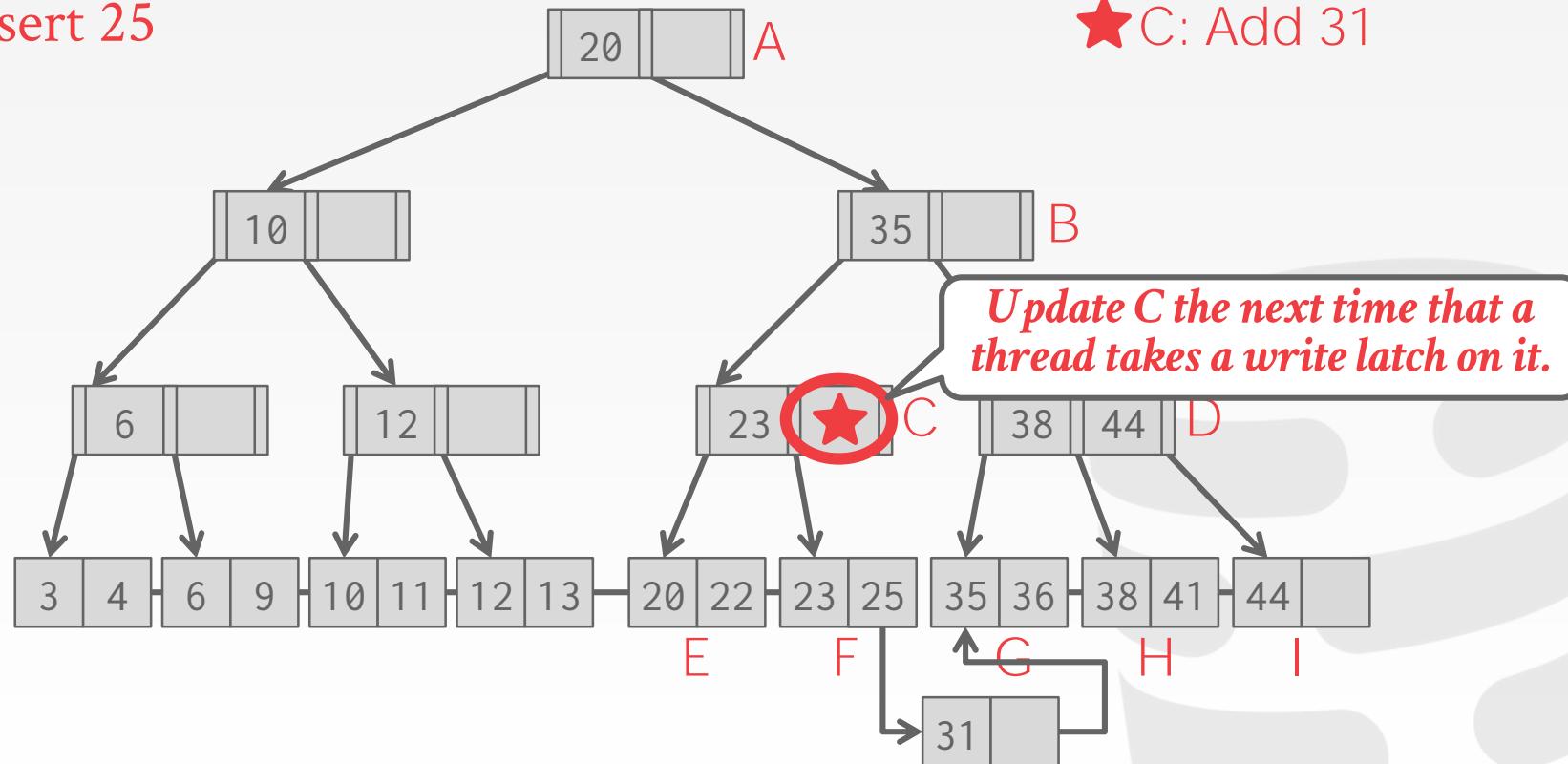
EXAMPLE #4 – INSERT 25

T₁: Insert 25



EXAMPLE #4 – INSERT 25

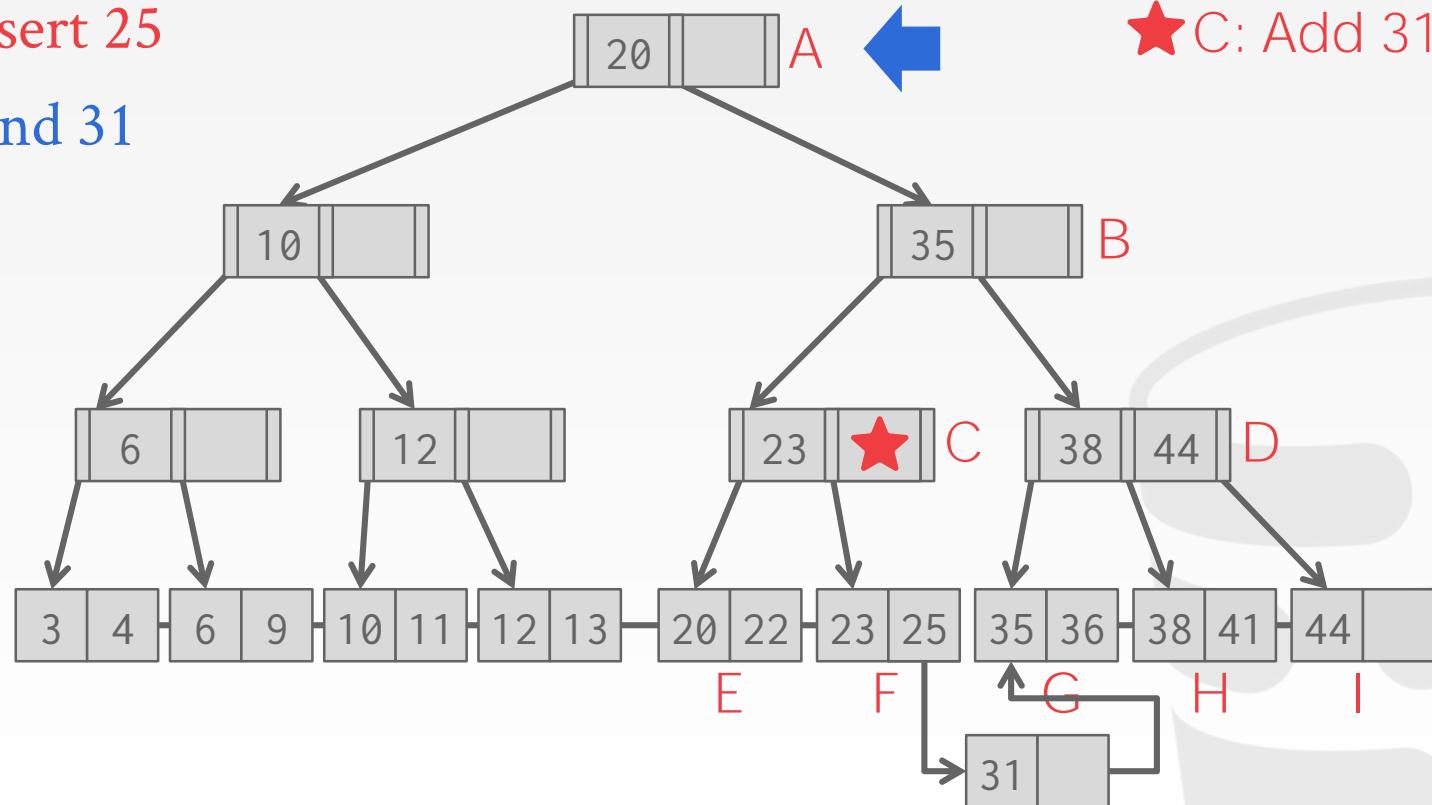
T₁: Insert 25



EXAMPLE #4 – INSERT 25

T₁: Insert 25

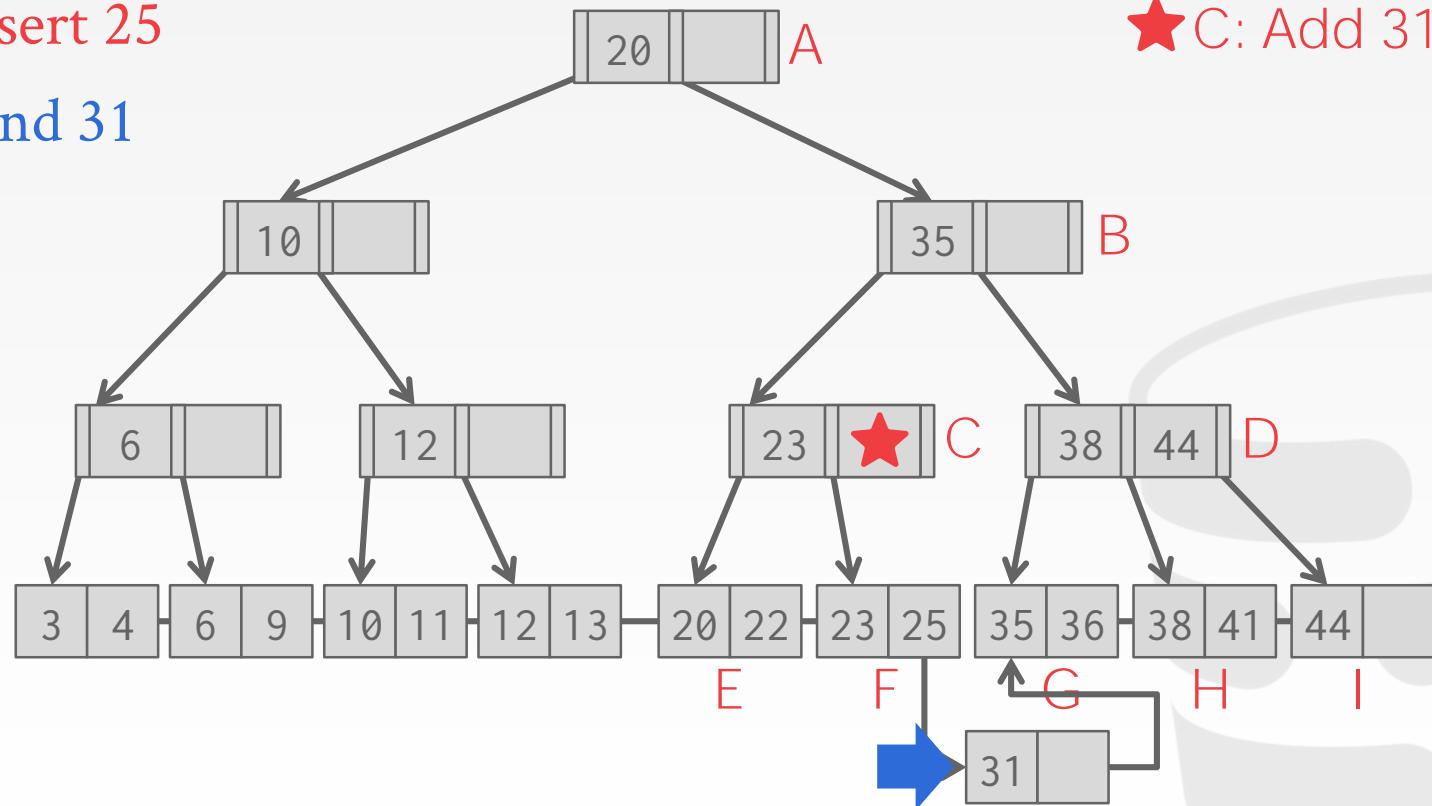
T₂: Find 31



EXAMPLE #4 – INSERT 25

T₁: Insert 25

T₂: Find 31

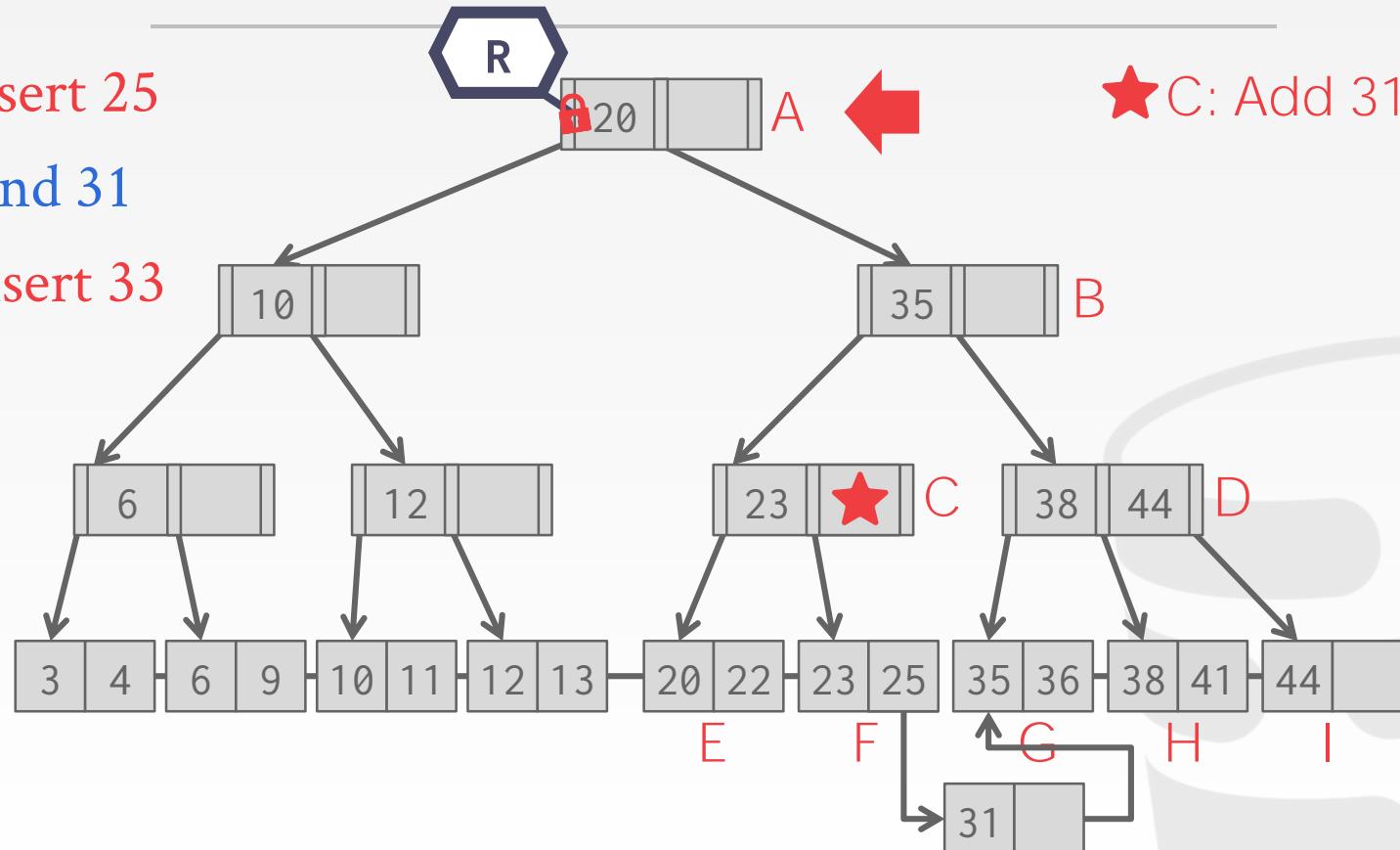


EXAMPLE #4 – INSERT 25

T₁: Insert 25

T₂: Find 31

T₃: Insert 33

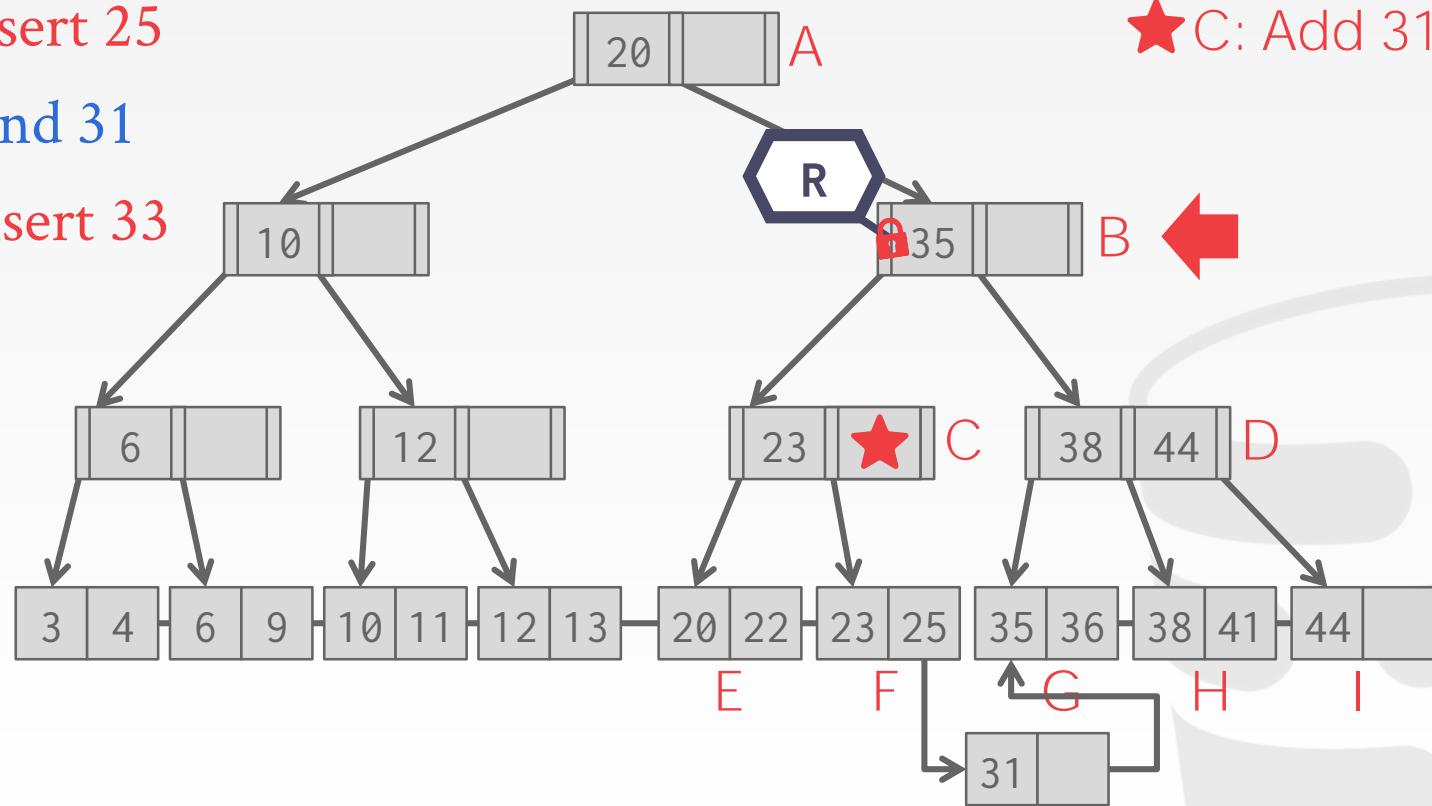


EXAMPLE #4 – INSERT 25

T₁: Insert 25

T₂: Find 31

T₃: Insert 33

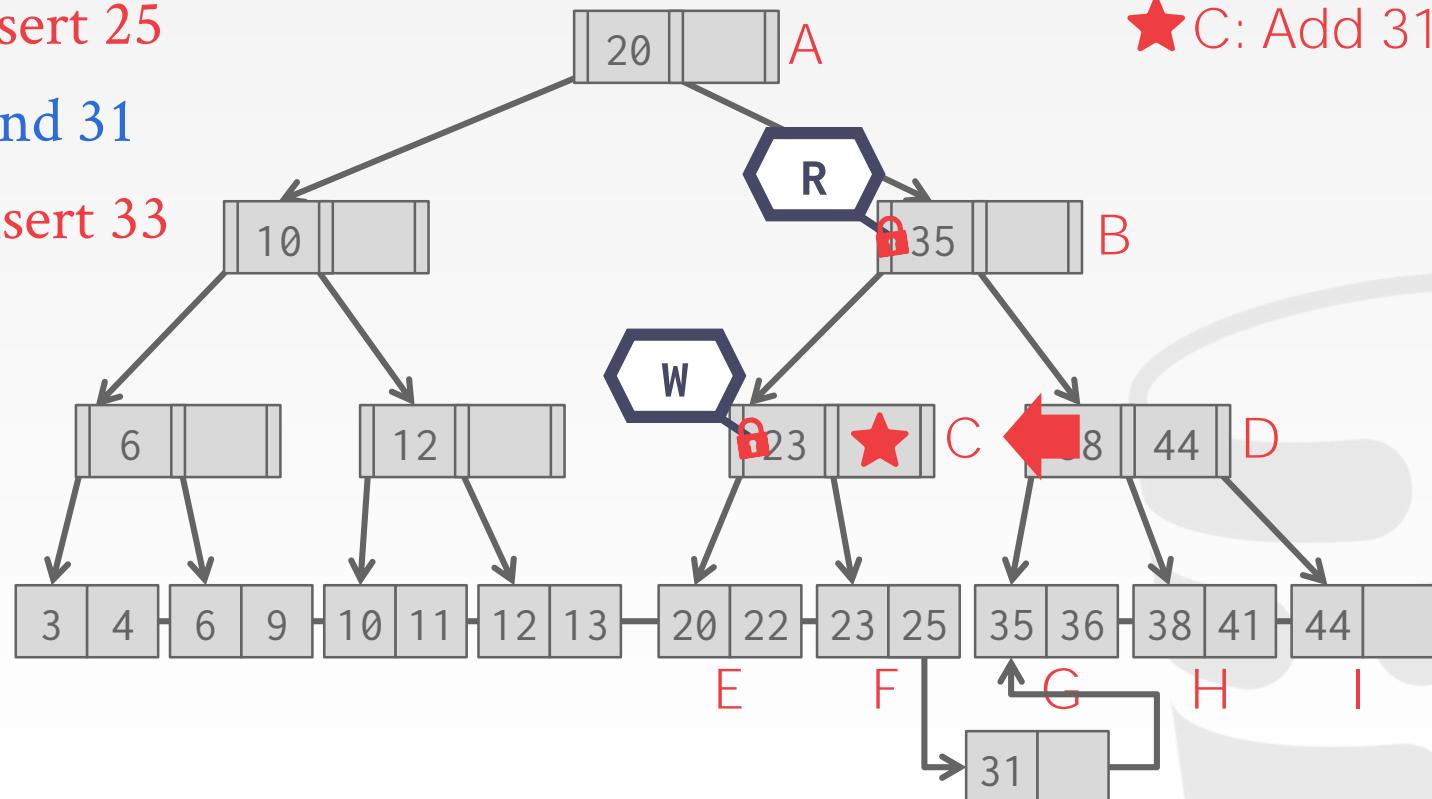


EXAMPLE #4 – INSERT 25

T₁: Insert 25

T₂: Find 31

T₃: Insert 33

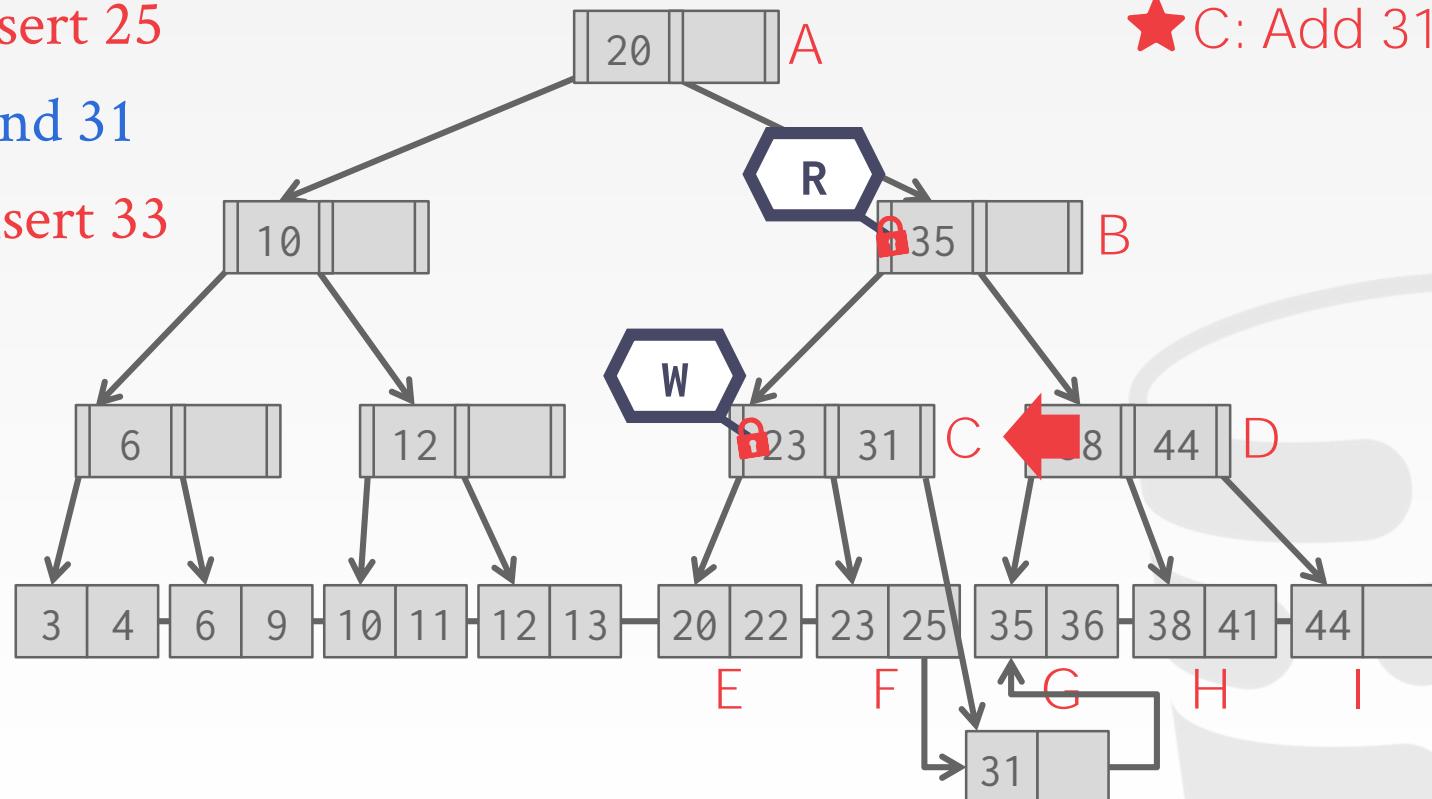


EXAMPLE #4 – INSERT 25

T₁: Insert 25

T₂: Find 31

T₃: Insert 33

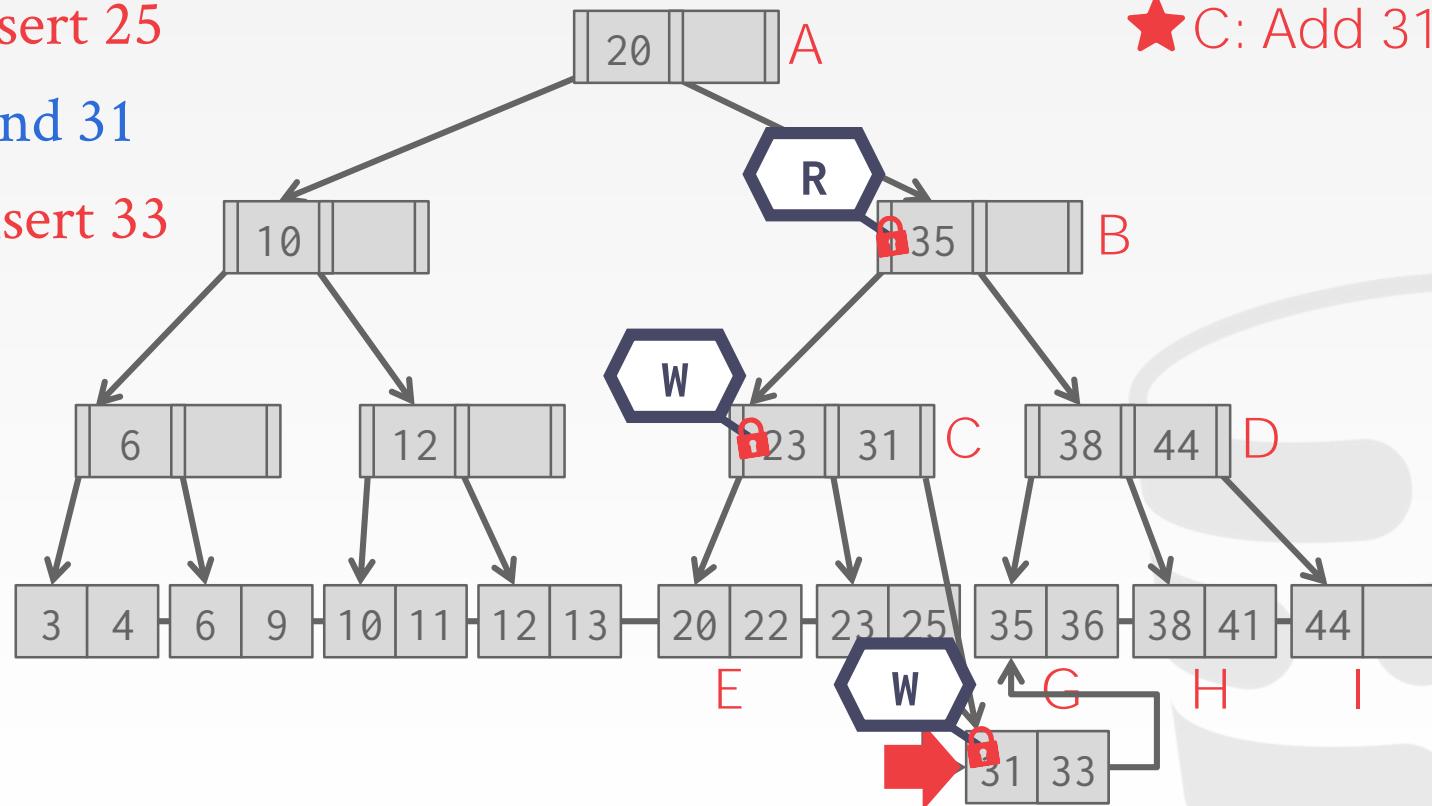


EXAMPLE #4 – INSERT 25

T₁: Insert 25

T₂: Find 31

T₃: Insert 33



CONCLUSION

Making a data structure thread-safe is notoriously difficult in practice.

We focused on B+Trees but the same high-level techniques are applicable to other data structures.



NEXT CLASS

We are finally going to discuss how to execute some queries...



10 |

Sorting & Aggregations



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #3 is due Wed Oct 9th @ 11:59pm

Mid-Term Exam is Wed Oct 16th @ 12:00pm

Project #2 is due Sun Oct 20th @ 11:59pm



COURSE STATUS

We are now going to talk about how to execute queries using table heaps and indexes.

Next two weeks:

- Operator Algorithms
- Query Processing Models
- Runtime Architectures

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

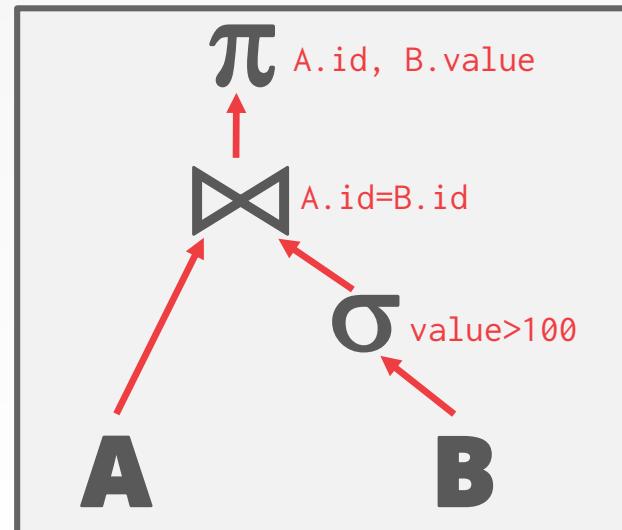
QUERY PLAN

The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

The output of the root node is the result of the query.

```
SELECT A.id, B.value
FROM A, B
WHERE A.id = B.id
AND B.value > 100
```

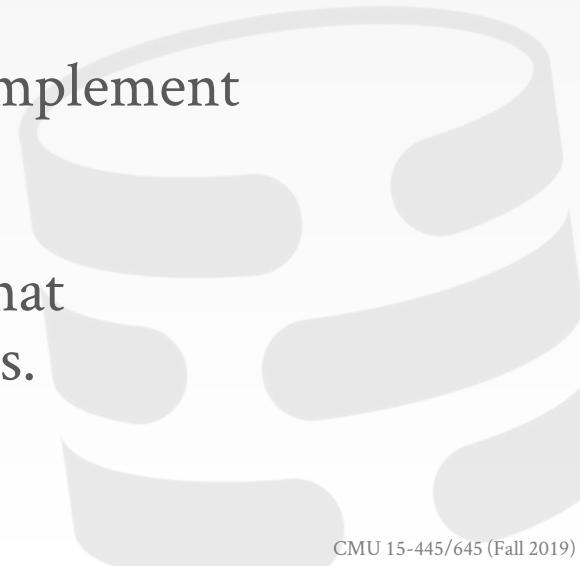


DISK-ORIENTED DBMS

Just like it cannot assume that a table fits entirely in memory, a disk-oriented DBMS cannot assume that the results of a query fits in memory.

We are going to use the buffer pool to implement algorithms that need to spill to disk.

We are also going to prefer algorithms that maximize the amount of sequential access.



TODAY'S AGENDA

External Merge Sort
Aggregations



WHY DO WE NEED TO SORT?

Tuples in a table have no specific order.

But queries often want to retrieve tuples in a specific order.

- Trivial to support duplicate elimination (**DISTINCT**).
- Bulk loading sorted tuples into a B+Tree index is faster.
- Aggregations (**GROUP BY**).

SORTING ALGORITHMS

If data fits in memory, then we can use a standard sorting algorithm like quick-sort.

If data does not fit in memory, then we need to use a technique that is aware of the cost of writing data out to disk...

EXTERNAL MERGE SORT

Divide-and-conquer sorting algorithm that splits the data set into separate runs and then sorts them individually.

Phase #1 – Sorting

- Sort blocks of data that fit in main-memory and then write back the sorted blocks to a file on disk.

Phase #2 – Merging

- Combine sorted sub-files into a single larger file.

2-WAY EXTERNAL MERGE SORT

We will start with a simple example of a 2-way external merge sort.

→ "2" represents the number of runs that we are going to merge into a new run for each pass.

Data set is broken up into ***N*** pages.

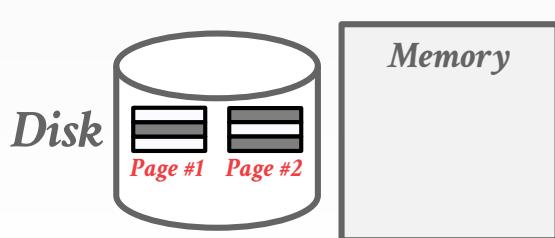
The DBMS has a finite number of ***B*** buffer pages to hold input and output data.



2-WAY EXTERNAL MERGE SORT

Pass #0

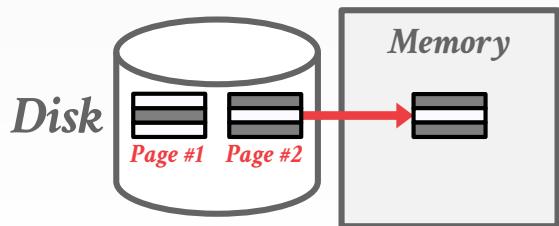
- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

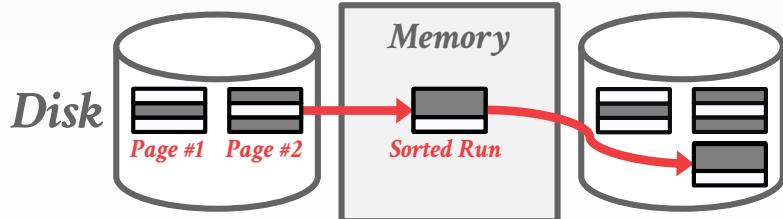
- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

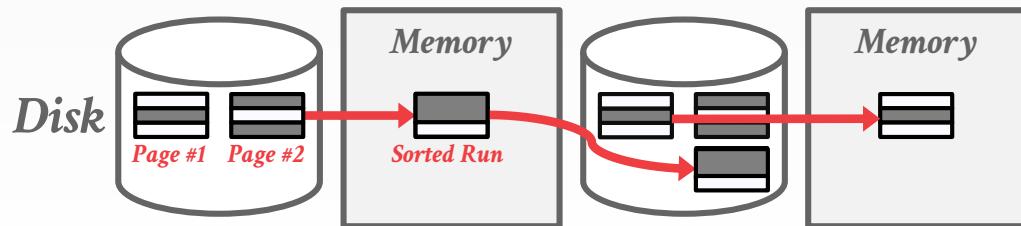
- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

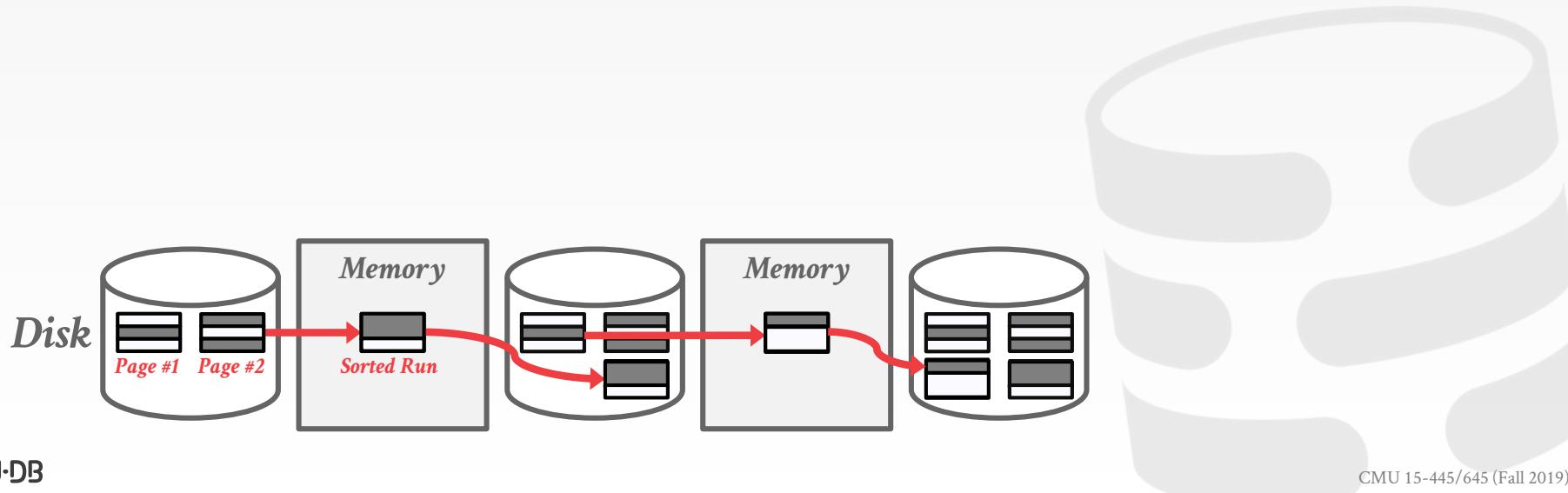
- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.



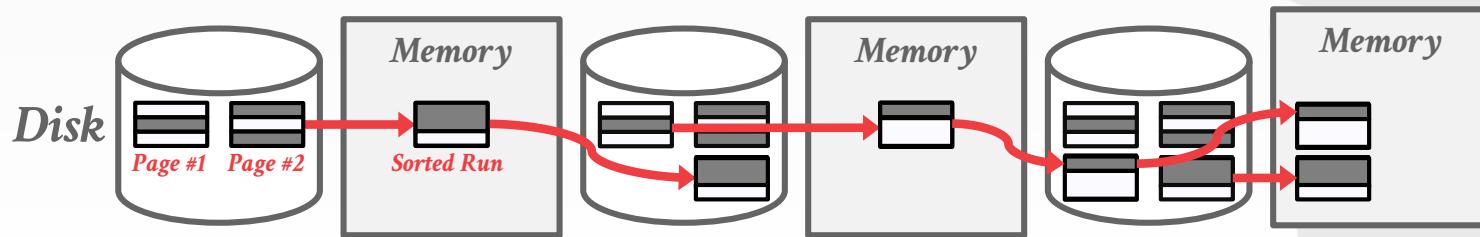
2-WAY EXTERNAL MERGE SORT

Pass #0

- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.

Pass #1,2,3,...

- Recursively merges pairs of runs into runs twice as long.
- Uses three buffer pages (2 for input pages, 1 for output).



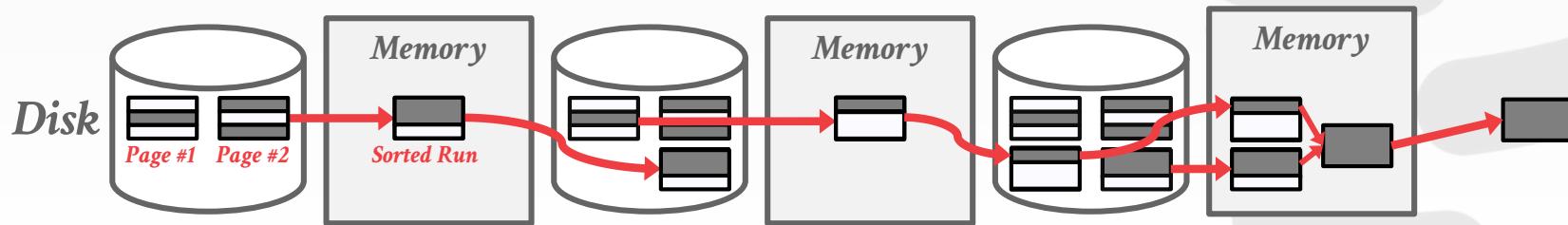
2-WAY EXTERNAL MERGE SORT

Pass #0

- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.

Pass #1,2,3,...

- Recursively merges pairs of runs into runs twice as long.
- Uses three buffer pages (2 for input pages, 1 for output).



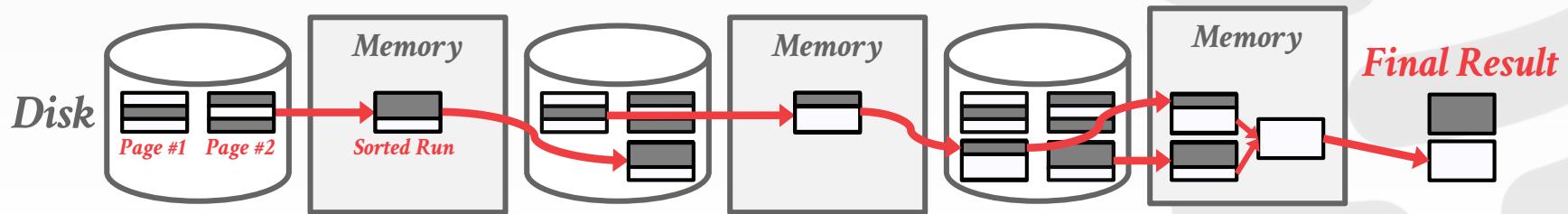
2-WAY EXTERNAL MERGE SORT

Pass #0

- Read every B pages of the table into memory
- Sort pages into runs and write them back to disk.

Pass #1,2,3,...

- Recursively merges pairs of runs into runs twice as long.
- Uses three buffer pages (2 for input pages, 1 for output).



2-WAY EXTERNAL MERGE SORT

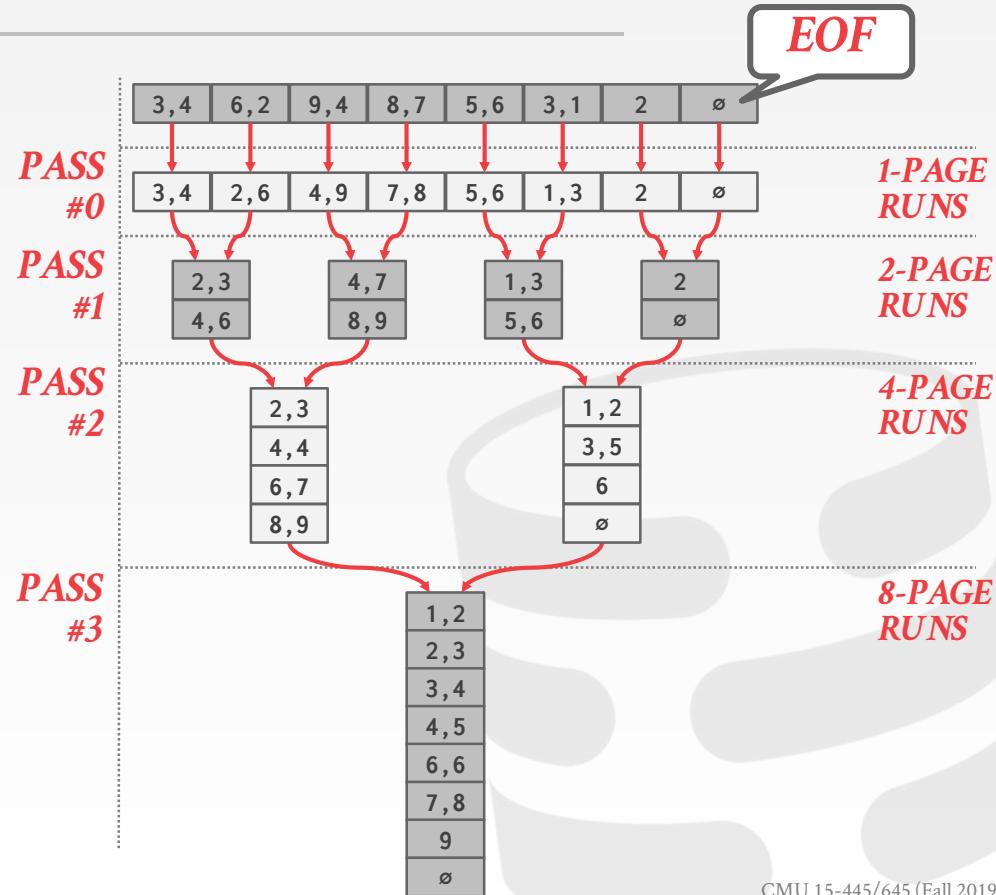
In each pass, we read and write each page in file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



2-WAY EXTERNAL MERGE SORT

This algorithm only requires three buffer pages to perform the sorting ($B=3$).

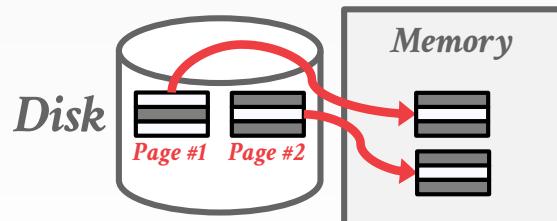
But even if we have more buffer space available ($B>3$), it does not effectively utilize them...



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

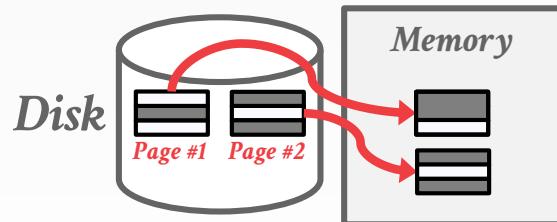
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



GENERAL EXTERNAL MERGE SORT

Pass #0

- Use B buffer pages.
- Produce $\lceil N / B \rceil$ sorted runs of size B

Pass #1,2,3,...

- Merge $B-1$ runs (i.e., K-way merge).

Number of passes = $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

Total I/O Cost = $2N \cdot (\# \text{ of passes})$



GENERAL EXTERNAL MERGE SORT

Pass #0

- Use B buffer pages.
- Produce $\lceil N / B \rceil$ sorted runs of size B

Pass #1,2,3,...

- Merge $B-1$ runs (i.e., K-way merge).

Number of passes = $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

Total I/O Cost = $2N \cdot (\# \text{ of passes})$



EXAMPLE

Sort 108 pages with 5 buffer pages: $N=108, B=5$

- **Pass #0:** $\lceil N / B \rceil = \lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages).
- **Pass #1:** $\lceil N' / B-1 \rceil = \lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages).
- **Pass #2:** $\lceil N'' / B-1 \rceil = \lceil 6 / 4 \rceil = 2$ sorted runs, first one has 80 pages and second one has 28 pages.
- **Pass #3:** Sorted file of 108 pages.

$$\begin{aligned}1 + \lceil \log_{B-1}[N / B] \rceil &= 1 + \lceil \log_4 22 \rceil = 1 + \lceil 2.229... \rceil \\&= 4 \text{ passes}\end{aligned}$$

USING B+TREES FOR SORTING

If the table that must be sorted already has a B+Tree index on the sort attribute(s), then we can use that to accelerate sorting.

Retrieve tuples in desired sort order by simply traversing the leaf pages of the tree.

Cases to consider:

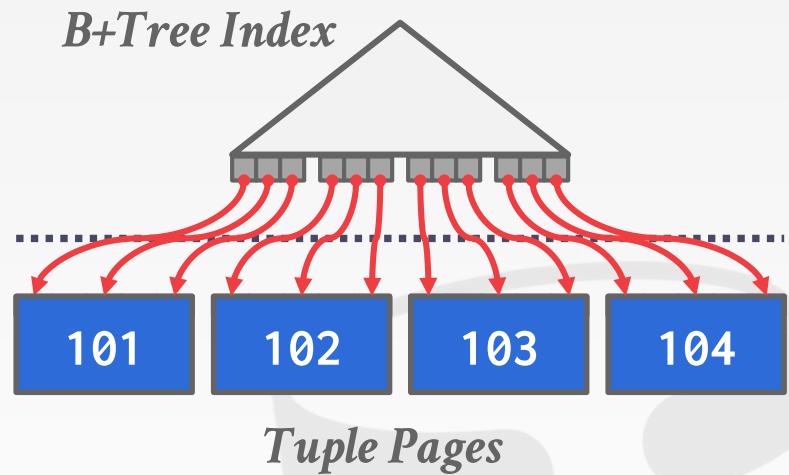
- Clustered B+Tree
- Unclustered B+Tree



CASE #1 – CLUSTERED B+TREE

Traverse to the left-most leaf page,
and then retrieve tuples from all leaf
pages.

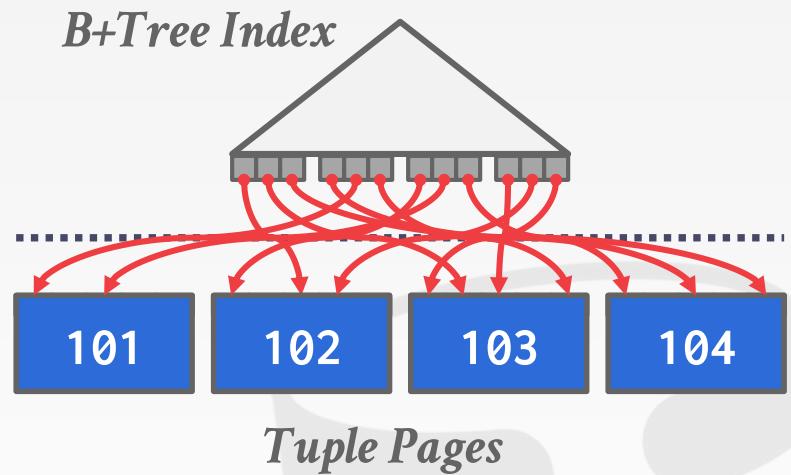
This is always better than external
sorting because there is no
computational cost and all disk access
is sequential.



CASE #2 – UNCLUSTERED B+TREE

Chase each pointer to the page that contains the data.

This is almost always a bad idea.
In general, one I/O per data record.



AGGREGATIONS

Collapse multiple tuples into a single scalar value.

Two implementation choices:

- Sorting
- Hashing



SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
 ORDER BY cid
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

**Remove
Columns**

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
15-445
15-721
15-826

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
 ORDER BY cid
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove
Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
15-445
15-721
15-826

Eliminate
Dups

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
 ORDER BY cid
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove
Columns

cid
15-445
15-826
15-721
15-445

Sort

Eliminate Dups

cid
15-445
15-826
15-721
15-445

ALTERNATIVES TO SORTING

What if we don't need the data to be ordered?

- Forming groups in **GROUP BY** (no ordering)
- Removing duplicates in **DISTINCT** (no ordering)

Hashing is a better alternative in this scenario.

- Only need to remove duplicates, no need for ordering.
- Can be computationally cheaper than sorting.

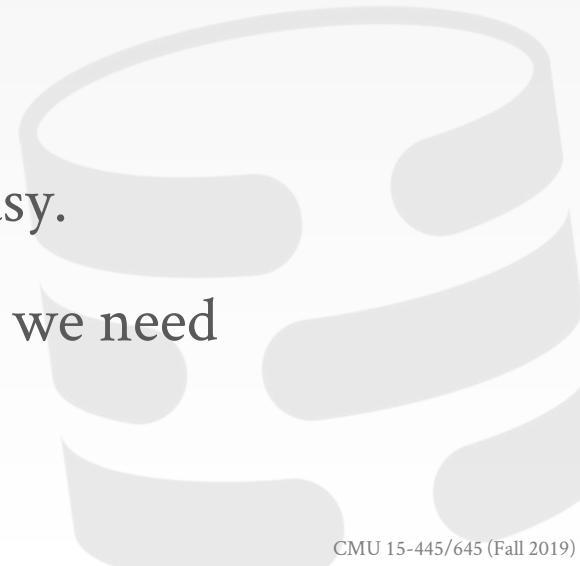
HASHING AGGREGATE

Populate an ephemeral hash table as the DBMS scans the table. For each record, check whether there is already an entry in the hash table:

- **DISTINCT**: Discard duplicate.
- **GROUP BY**: Perform aggregate computation.

If everything fits in memory, then it is easy.

If the DBMS must spill data to disk, then we need to be smarter...



EXTERNAL HASHING AGGREGATE

Phase #1 – Partition

- Divide tuples into buckets based on hash key.
- Write them out to disk when they get full.

Phase #2 – ReHash

- Build in-memory hash table for each partition and compute the aggregation.



PHASE #1 – PARTITION

Use a hash function h_1 to split tuples into partitions on disk.

- We know that all matches live in the same partition.
- Partitions are "spilled" to disk via output buffers.

Assume that we have B buffers.

We will use $B-1$ buffers for the partitions and 1 buffer for the input data.



PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

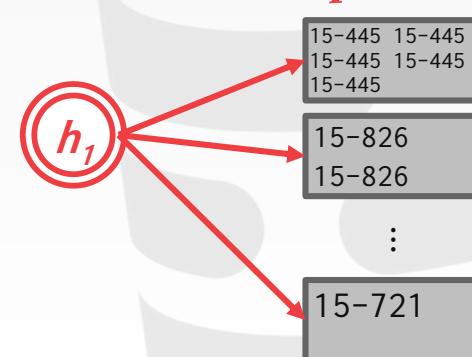
sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

**Remove
Columns**

cid
15-445
15-826
15-721
15-445

⋮

B-1 partitions



PHASE #2 – REHASH

For each partition on disk:

- Read it into memory and build an in-memory hash table based on a second hash function h_2 .
- Then go through each bucket of this hash table to bring together matching tuples.

This assumes that each partition fits in memory.

PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets

15-445 15-445
15-445 15-445
15-445

15-826
15-826

:

PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
```

Phase #1 Buckets

15-445	15-445
15-445	15-445
15-445	

15-826
15-826

:

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

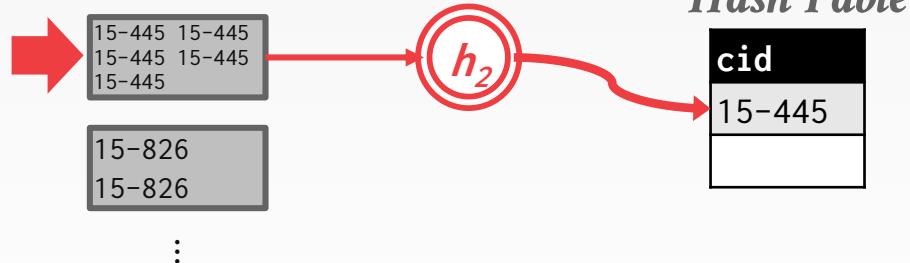
PHASE #2 – REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



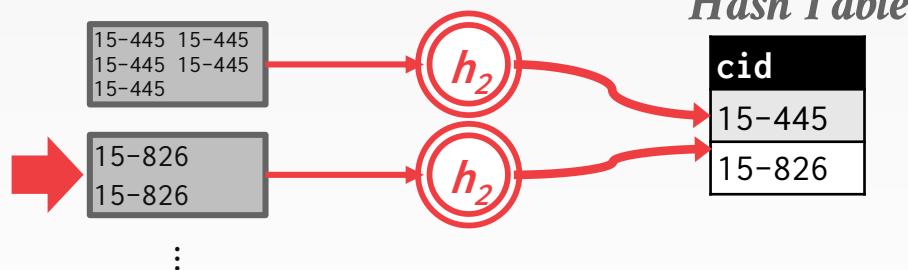
PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B', 'C')
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



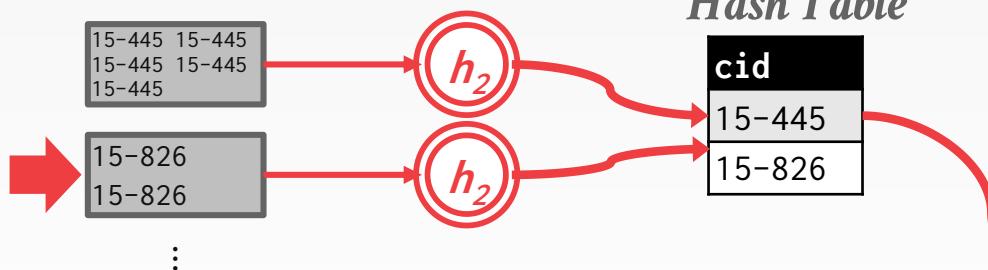
PHASE #2 – REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



Hash Table

Final Result

cid
15-445
15-826

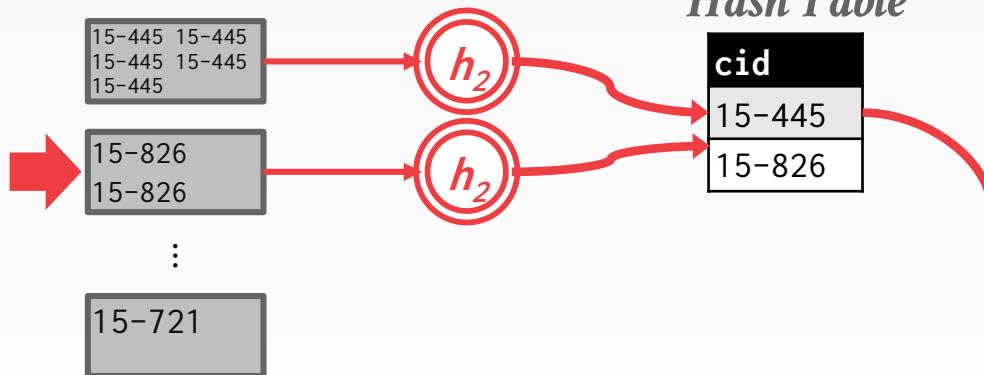
PHASE #2 – REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



Final Result

cid
15-445
15-826

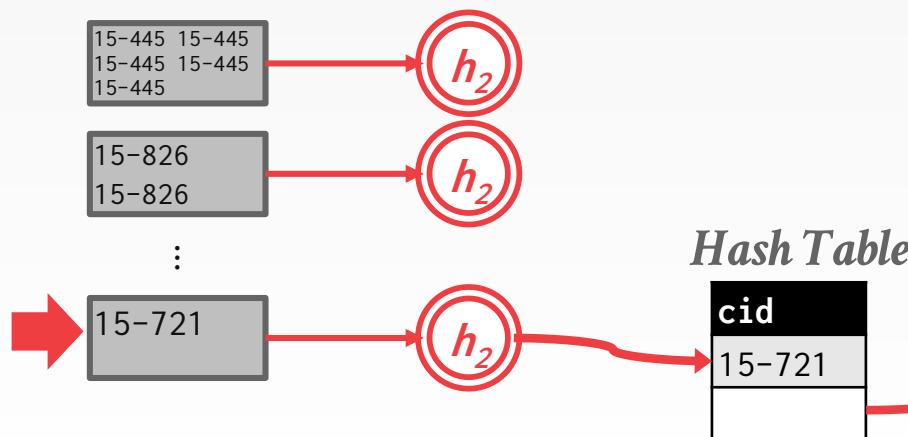
PHASE #2 – REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

enrolled(sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



Final Result

cid
15-445
15-826
15-721

HASHING SUMMARIZATION

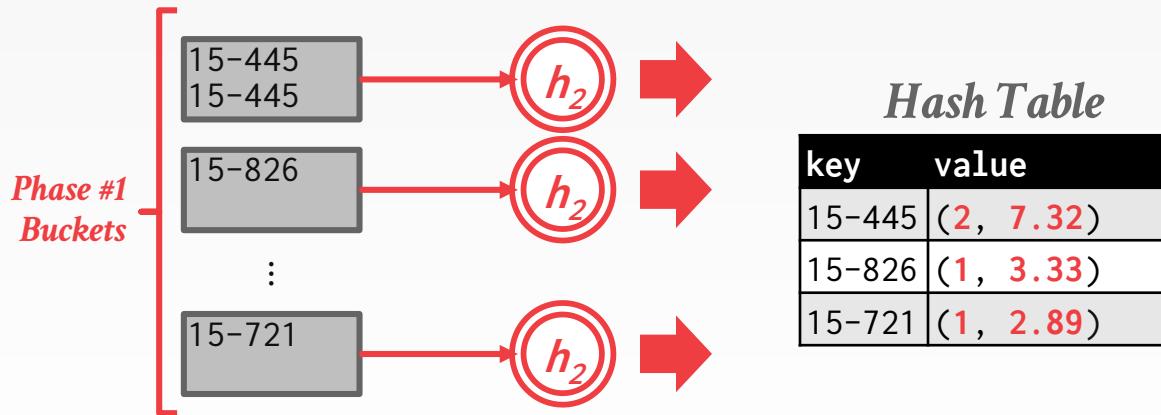
During the ReHash phase, store pairs of the form
(GroupKey→RunningVal)

When we want to insert a new tuple into the hash table:

- If we find a matching **GroupKey**, just update the **RunningVal** appropriately
- Else insert a new **GroupKey→RunningVal**

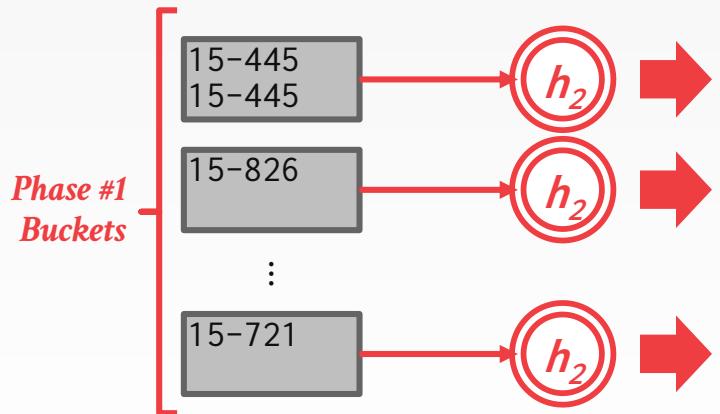
HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
GROUP BY cid
```



HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
GROUP BY cid
```

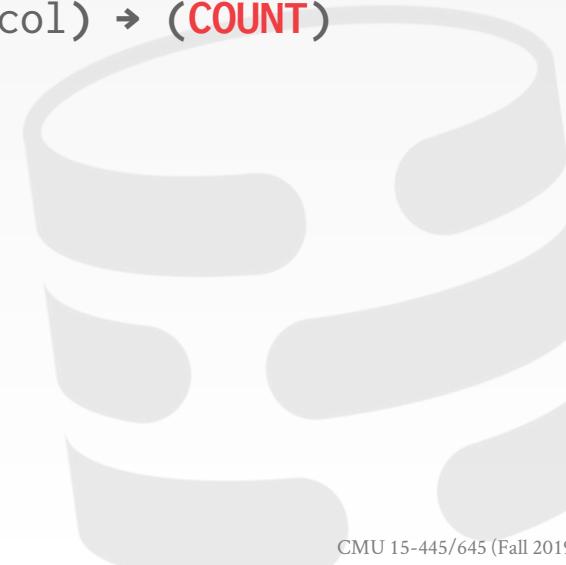


Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

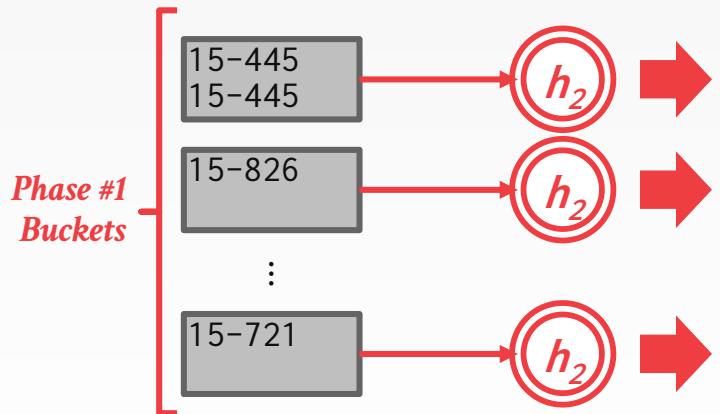
Running Totals

- $\text{AVG}(\text{col}) \rightarrow (\text{COUNT}, \text{SUM})$
- $\text{MIN}(\text{col}) \rightarrow (\text{MIN})$
- $\text{MAX}(\text{col}) \rightarrow (\text{MAX})$
- $\text{SUM}(\text{col}) \rightarrow (\text{SUM})$
- $\text{COUNT}(\text{col}) \rightarrow (\text{COUNT})$



HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
GROUP BY cid
```



Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Running Totals

- $\text{AVG}(\text{col}) \rightarrow (\text{COUNT}, \text{SUM})$
- $\text{MIN}(\text{col}) \rightarrow (\text{MIN})$
- $\text{MAX}(\text{col}) \rightarrow (\text{MAX})$
- $\text{SUM}(\text{col}) \rightarrow (\text{SUM})$
- $\text{COUNT}(\text{col}) \rightarrow (\text{COUNT})$

Final Result

cid	AVG(gpa)
15-445	3.66
15-826	3.33
15-721	2.89

COST ANALYSIS

How big of a table can we hash using this approach?

- $B-1$ "spill partitions" in Phase #1
- Each should be no more than B blocks big

Answer: $B \cdot (B-1)$

- A table of N pages needs about \sqrt{N} buffers
- Assumes hash distributes records evenly.
Use a "fudge factor" $f > 1$ for that: we need
 $B \cdot \sqrt{f \cdot N}$



CONCLUSION

Choice of sorting vs. hashing is subtle and depends on optimizations done in each case.

We already discussed the optimizations for sorting:

- Chunk I/O into large blocks to amortize seek+RD costs.
- Double-buffering to overlap CPU and I/O.

PROJECT #2

You will build a thread-safe linear probing hash table that supports automatic resizing.

We define the API for you. You need to provide the implementation.



<https://15445.courses.cs.cmu.edu/fall2019/project2/>

PROJECT #2 – TASKS

Page Layouts

Hash Table Implementation

Table Resizing

Concurrency Control Protocol



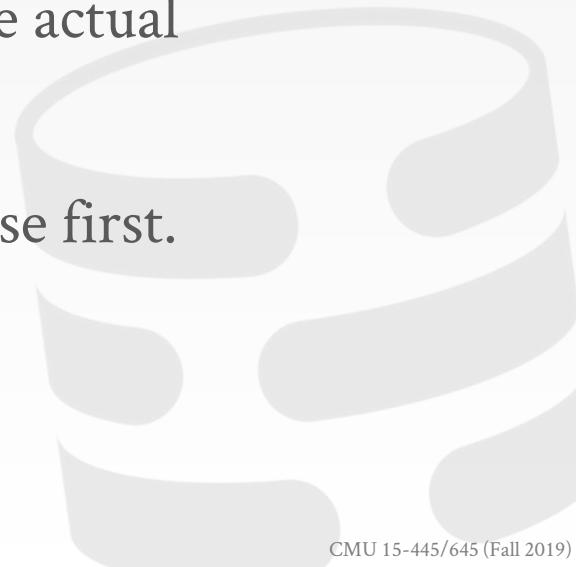
DEVELOPMENT HINTS

Follow the textbook semantics and algorithms.

You should make sure your page layout are working correctly before switching to the actual hash table itself.

Then focus on the single-threaded use case first.

Avoid premature optimizations.
→ Correctness first, performance second.



THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Rebase on top of the latest BusTub master branch.

Post your questions on Piazza or come to TA office hours.

PLAGIARISM WARNING

Your project implementation must be your own work.

- You may not copy source code from other groups or the web.
- Do not publish your implementation on Github.

Plagiarism will not be tolerated.

See [CMU's Policy on Academic Integrity](#) for additional information.



NEXT CLASS

Nested Loop Join

Sort-Merge Join

Hash Join



11

Join Algorithms



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

WHY DO WE NEED TO JOIN?

We normalize tables in a relational database to avoid unnecessary repetition of information.

We use the join operate to reconstruct the original tuples without any information loss.



JOIN ALGORITHMS

We will focus on combining two tables at a time with inner equijoin algorithms.

→ These algorithms can be tweaked to support other joins.

In general, we want the smaller table to always be the left table ("outer table") in the query plan.

JOIN OPERATORS

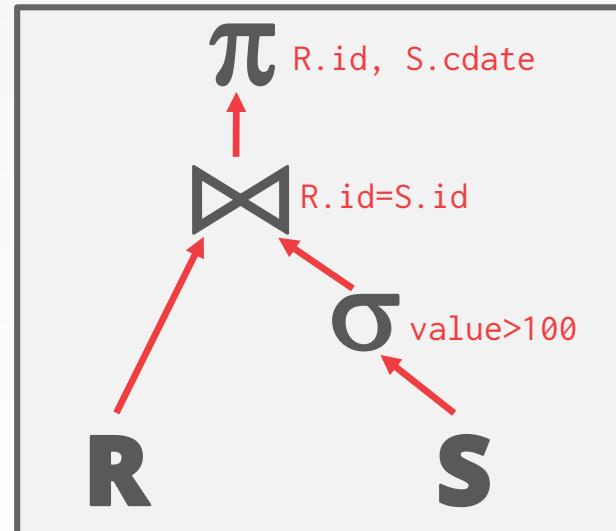
Decision #1: Output

- What data does the join operator emit to its parent operator in the query plan tree?

Decision #2: Cost Analysis Criteria

- How do we determine whether one join algorithm is better than another?

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



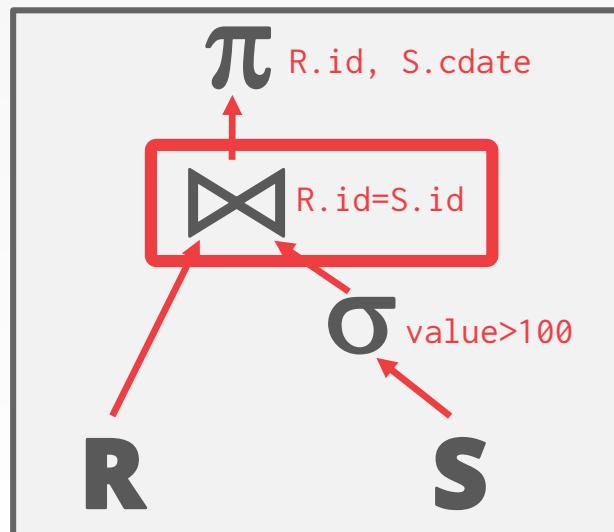
OPERATOR OUTPUT

For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, concatenate r and s together into a new tuple.

Contents can vary:

- Depends on processing model
- Depends on storage model
- Depends on the query

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

```
SELECT R.id, S.cdate
  FROM R JOIN S
  ON R.id = S.id
 WHERE S.value > 100
```

R(id, name) S(id, value, cdate)

id	name		id	value	cdate
123	abc		123	1000	10/2/2019
			123	2000	10/2/2019

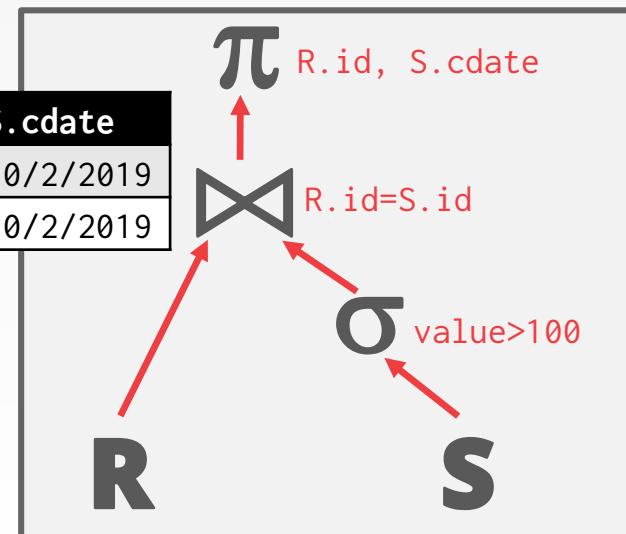
R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/2/2019
123	abc	123	2000	10/2/2019

JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/2/2019
123	abc	123	2000	10/2/2019

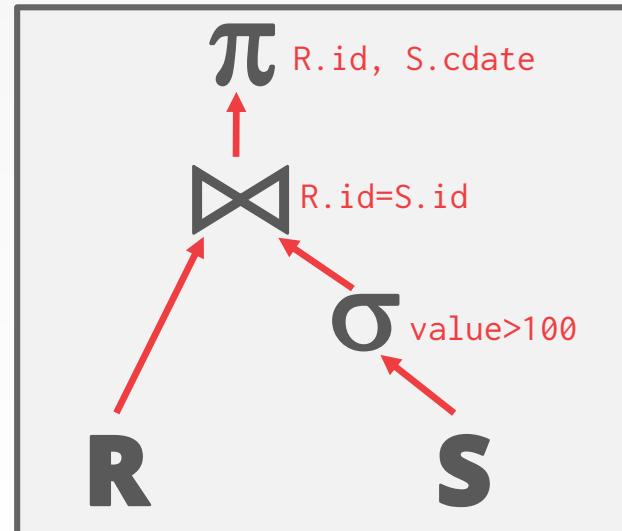


JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

Subsequent operators in the query plan never need to go back to the base tables to get more data.

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

```
SELECT R.id, S.cdate
  FROM R JOIN S
  ON R.id = S.id
 WHERE S.value > 100
```

R(id, name) S(id, value, cdate)

id	name		id	value	cdate
123	abc		123	1000	10/2/2019
			123	2000	10/2/2019

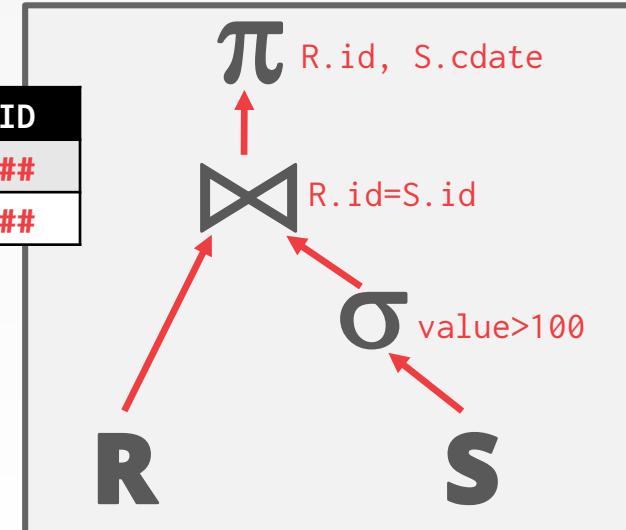
R.id	R.RID	S.id	S.RID
123	R.###	123	S.###
123	R.###	123	S.###

OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

R.id	R.RID	S.id	S.RID
123	R.###	123	S.###
123	R.###	123	S.###



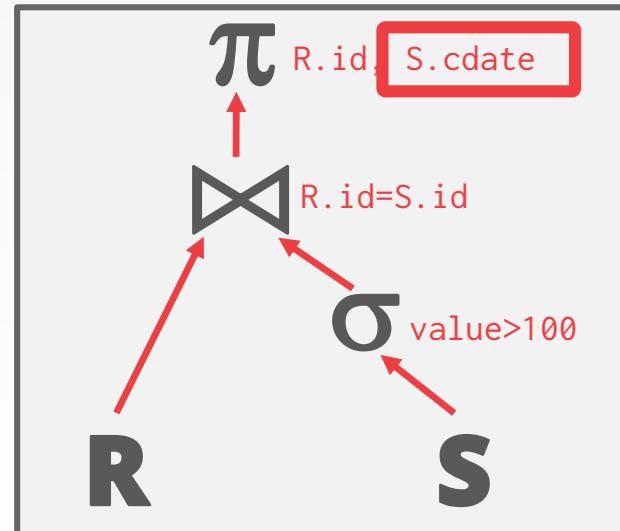
OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

Ideal for column stores because the DBMS does not copy data that is not needed for the query.

This is called **late materialization**.

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



I/O COST ANALYSIS

Assume:

- M pages in table **R**, m tuples in **R**
- N pages in table **S**, n tuples in **S**

```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```

Cost Metric: # of IOs to compute join

We will ignore output costs since that depends on the data and we cannot compute that yet.

JOIN VS CROSS-PRODUCT

RXS is the most common operation and thus must be carefully optimized.

RxS followed by a selection is inefficient because the cross-product is large.

There are many algorithms for reducing join cost, but no algorithm works well in all scenarios.



JOIN ALGORITHMS

Nested Loop Join

- Simple / Stupid
- Block
- Index

Sort-Merge Join

Hash Join

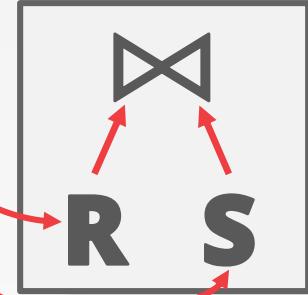


NESTED LOOP JOIN

```
foreach tuple r  $\in$  R:  

    foreach tuple s  $\in$  S:  

        emit, if r and s match
```



R(id, name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

STUPID NESTED LOOP JOIN

Why is this algorithm stupid?

→ For every tuple in **R**, it scans **S** once

Cost: $M + (m \cdot N)$

M pages
 m tuples

R(id, name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

N pages
 n tuples

STUPID NESTED LOOP JOIN

Example database:

- Table **R**: $M = 1000$, $m = 100,000$
 - Table **S**: $N = 500$, $n = 40,000$
- $\left.\begin{array}{l} \\ \end{array}\right\} 4 \text{ KB pages} \rightarrow 6 \text{ MB}$

Cost Analysis:

- $M + (m \cdot N) = 1000 + (100000 \cdot 500) = 50,001,000 \text{ IOs}$
- At 0.1 ms/IO, Total time ≈ 1.3 hours

What if smaller table (**S**) is used as the outer table?

- $N + (n \cdot M) = 500 + (40000 \cdot 1000) = 40,000,500 \text{ IOs}$
- At 0.1 ms/IO, Total time ≈ 1.1 hours

BLOCK NESTED LOOP JOIN

```

foreach block  $B_R \in R$ :
  foreach block  $B_S \in S$ :
    foreach tuple  $r \in B_R$ :
      foreach tuple  $s \in B_S$ :
        emit, if  $r$  and  $s$  match
  
```

$R(id, name)$

id	$name$
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
 m tuples

$S(id, value, cdate)$

id	$value$	$cdate$
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

N pages
 n tuples

BLOCK NESTED LOOP JOIN

This algorithm performs fewer disk accesses.

→ For every block in **R**, it scans **S** once

Cost: $M + (M \cdot N)$

The diagram illustrates the Block Nested Loop Join algorithm. It shows two tables, **R(id, name)** and **S(id, value, cdate)**. The table **R** has 8 tuples and is divided into 2 pages, with each page containing 4 tuples. The table **S** has 6 tuples and is divided into 2 pages, with each page containing 3 tuples. Red brackets on the left side of the tables indicate the number of pages and tuples per page. Red brackets on the right side of the tables indicate the number of pages and tuples per page.

R(id, name)	
id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id, value, cdate)		
id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

BLOCK NESTED LOOP JOIN

Which one should be the outer table?
 → The smaller table in terms of # of pages

The diagram illustrates the Block Nested Loop Join algorithm. It shows two tables, R and S, with their respective row counts and tuple counts.

R(id, name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

M pages
m tuples

N pages
n tuples

BLOCK NESTED LOOP JOIN

Example database:

- Table **R**: $M = 1000$, $m = 100,000$
- Table **S**: $N = 500$, $n = 40,000$

Cost Analysis:

- $M + (M \cdot N) = 1000 + (1000 \cdot 500) = 501,000 \text{ IOs}$
- At 0.1 ms/IO, Total time ≈ 50 seconds



BLOCK NESTED LOOP JOIN

What if we have B buffers available?

- Use $B-2$ buffers for scanning the outer table.
- Use one buffer for the inner table, one buffer for storing output.

$R(id, name)$

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

$S(id, value, cdate)$

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

M pages
 m tuples

N pages
 n tuples

BLOCK NESTED LOOP JOIN

```

foreach  $B - 2$  blocks  $b_R \in R$ :
    foreach block  $b_S \in S$ :
        foreach tuple  $r \in b_R$ :
            foreach tuple  $s \in b_S$ :
                emit, if  $r$  and  $s$  match

```

$R(id, name)$

id	$name$
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
 m tuples

$S(id, value, cdate)$

id	$value$	$cdate$
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

N pages
 n tuples

BLOCK NESTED LOOP JOIN

This algorithm uses $B-2$ buffers for scanning R .

Cost: $M + (\lceil M / (B-2) \rceil \cdot N)$

What if the outer relation completely fits in memory ($B > M+2$)?

- **Cost: $M + N = 1000 + 500 = 1500$ IOs**
- At 0.1ms/IO, Total time ≈ 0.15 seconds



NESTED LOOP JOIN

Why do basic nested loop joins suck?

- For each tuple in the outer table, we must do a sequential scan to check for a match in the inner table.

We can avoid sequential scans by using an index to find inner table matches.

- Use an existing index for the join.
- Build one on the fly (hash table, B+Tree).

INDEX NESTED LOOP JOIN

```

foreach tuple  $r \in R$ :
    foreach tuple  $s \in \text{Index}(r_i = s_j)$ :
        emit, if  $r$  and  $s$  match

```

M pages
 m tuples

R(id, name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

Index(S.id)



N pages
 n tuples

INDEX NESTED LOOP JOIN

Assume the cost of each index probe is some constant C per tuple.

Cost: $M + (m \cdot C)$

R(id, name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
 m tuples

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

Index(S.id)



N pages
 n tuples

NESTED LOOP JOIN

Pick the smaller table as the outer table.

Buffer as much of the outer table in memory as possible.

Loop over the inner table or use an index.



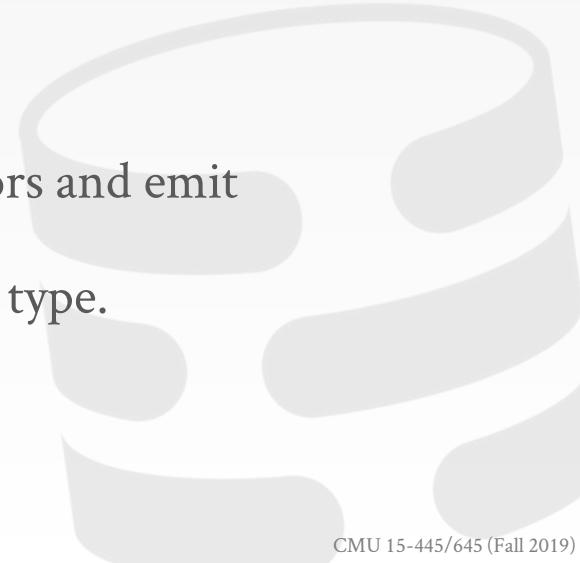
SORT-MERGE JOIN

Phase #1: Sort

- Sort both tables on the join key(s).
- We can use the external merge sort algorithm that we talked about last class.

Phase #2: Merge

- Step through the two sorted tables with cursors and emit matching tuples.
- May need to backtrack depending on the join type.



SORT-MERGE JOIN

```
sort R, S on join keys
cursorR ← Rsorted, cursorS ← Ssorted
while cursorR and cursorS:
    if cursorR > cursorS:
        increment cursorS
    if cursorR < cursorS:
        increment cursorR
    elif cursorR and cursorS match:
        emit
        increment cursorS
```

SORT-MERGE JOIN

R(id, name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
200	GZA
400	Raekwon

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
500	7777	10/2/2019
400	6666	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

Sort!



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

Sort!



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id,value,cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019
400	Raekwon	200	6666	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019
400	Raekwon	200	6666	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019
400	Raekwon	200	6666	10/2/2019
500	RZA	500	7777	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

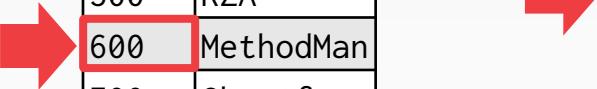
Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019
400	Raekwon	200	6666	10/2/2019
500	RZA	500	7777	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface



S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019
400	Raekwon	200	6666	10/2/2019
500	RZA	500	7777	10/2/2019

SORT-MERGE JOIN

R(id, name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)

id	value	cdate
100	2222	10/2/2019
100	9999	10/2/2019
200	8888	10/2/2019
400	6666	10/2/2019
500	7777	10/2/2019

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/2/2019
100	Andy	100	9999	10/2/2019
200	GZA	200	8888	10/2/2019
200	GZA	200	8888	10/2/2019
400	Raekwon	200	6666	10/2/2019
500	RZA	500	7777	10/2/2019

SORT-MERGE JOIN

Sort Cost (**R**): $2M \cdot (1 + \lceil \log_{B-1} [M / B] \rceil)$

Sort Cost (**S**): $2N \cdot (1 + \lceil \log_{B-1} [N / B] \rceil)$

Merge Cost: $(M + N)$

Total Cost: Sort + Merge



SORT-MERGE JOIN

Example database:

- Table **R**: $M = 1000$, $m = 100,000$
- Table **S**: $N = 500$, $n = 40,000$

With $B=100$ buffer pages, both **R** and **S** can be sorted in two passes:

- Sort Cost (**R**) = $2000 \cdot (1 + \lceil \log_{99} 1000 / 100 \rceil) = 4000$ IOs
- Sort Cost (**S**) = $1000 \cdot (1 + \lceil \log_{99} 500 / 100 \rceil) = 2000$ IOs
- Merge Cost = $(1000 + 500) = 1500$ IOs
- Total Cost = $4000 + 2000 + 1500 = 7500$ IOs
- At 0.1 ms/IO, Total time ≈ 0.75 seconds

SORT-MERGE JOIN

The worst case for the merging phase is when the join attribute of all of the tuples in both relations contain the same value.

Cost: $(M \cdot N) + (\text{sort cost})$



WHEN IS SORT-MERGE JOIN USEFUL?

One or both tables are already sorted on join key.

Output must be sorted on join key.

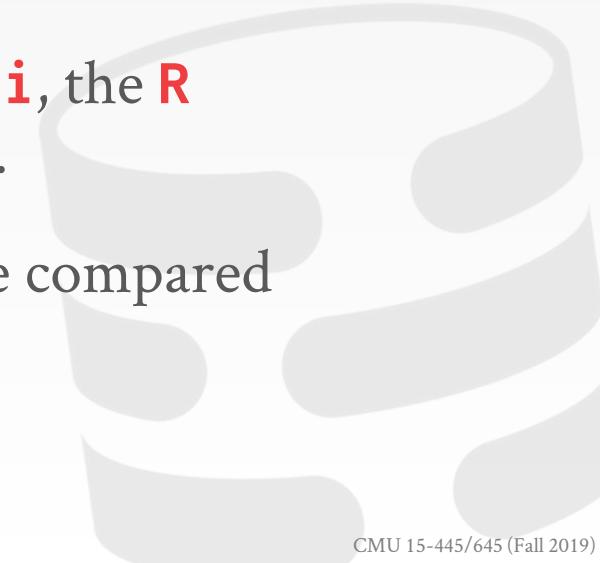
The input relations may be sorted by either by an explicit sort operator, or by scanning the relation using an index on the join key.

HASH JOIN

If tuple $r \in R$ and a tuple $s \in S$ satisfy the join condition, then they have the same value for the join attributes.

If that value is hashed to some partition i , the R tuple must be in r_i and the S tuple in s_i .

Therefore, R tuples in r_i need only to be compared with S tuples in s_i .



BASIC HASH JOIN ALGORITHM

Phase #1: Build

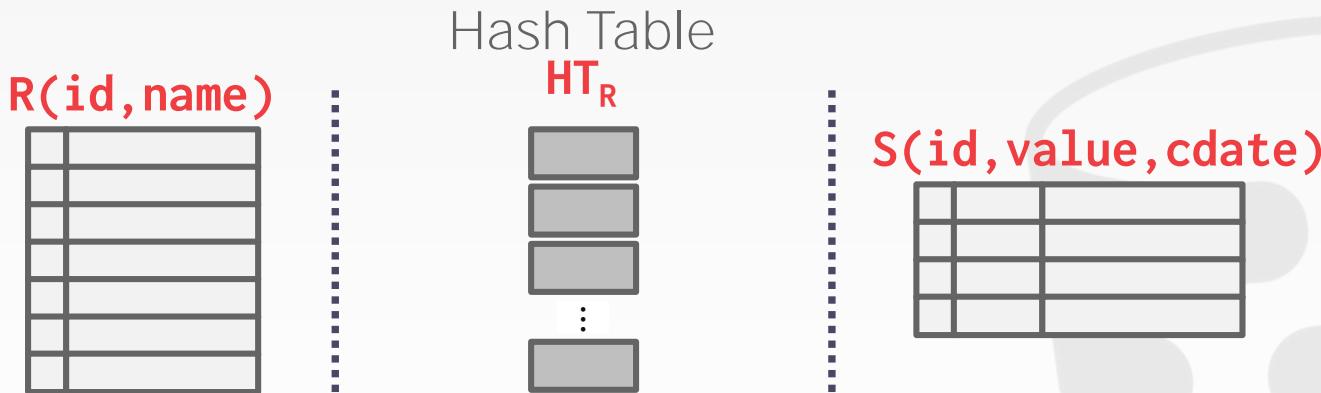
- Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

Phase #2: Probe

- Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

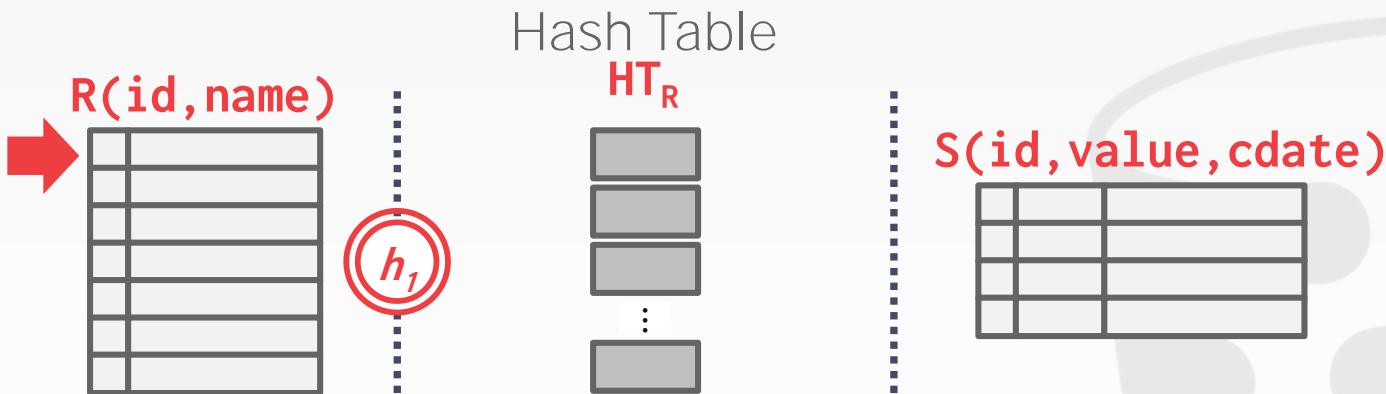
BASIC HASH JOIN ALGORITHM

```
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
    output, if  $h_1(s) \in HT_R$ 
```



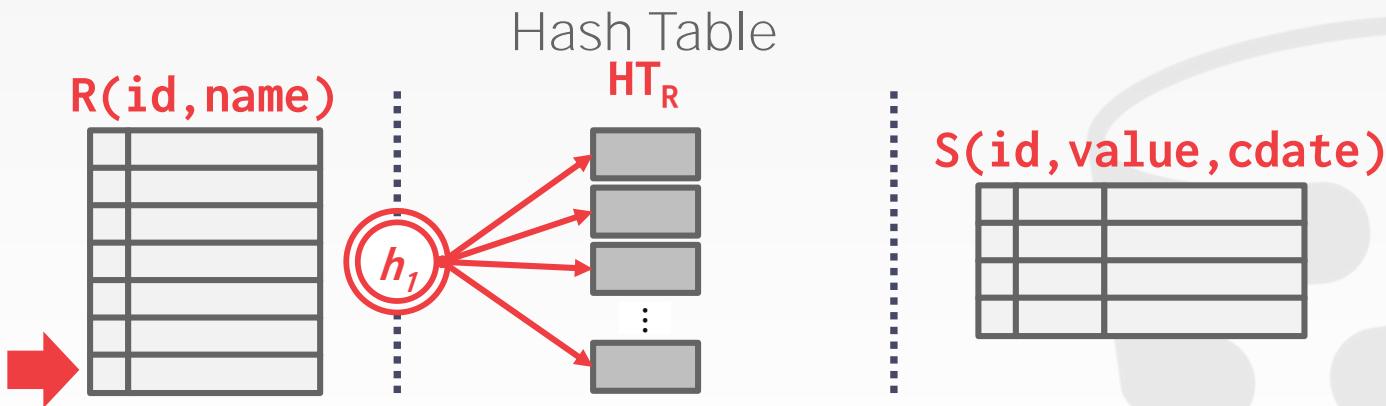
BASIC HASH JOIN ALGORITHM

```
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
    output, if  $h_1(s) \in HT_R$ 
```



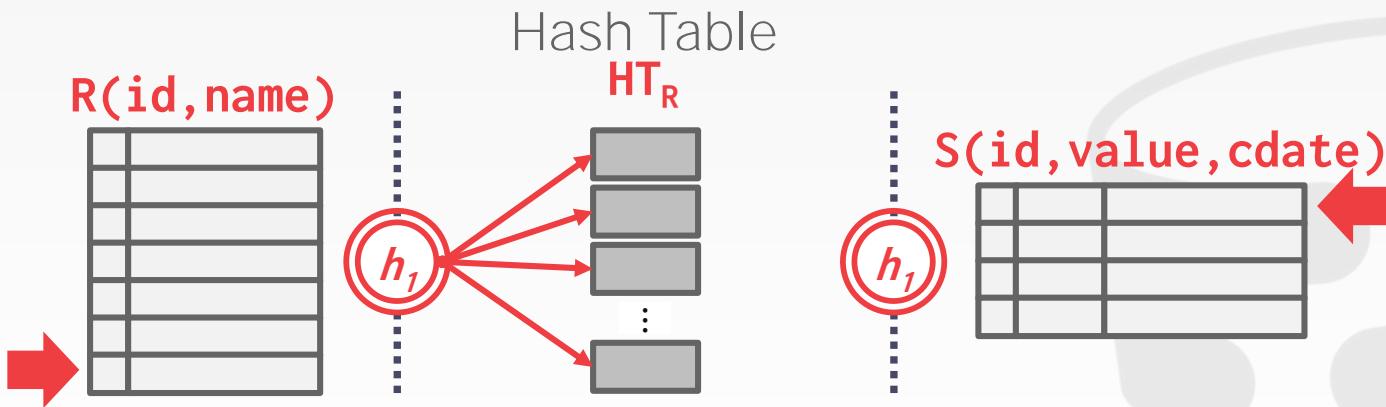
BASIC HASH JOIN ALGORITHM

```
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
    output, if  $h_1(s) \in HT_R$ 
```



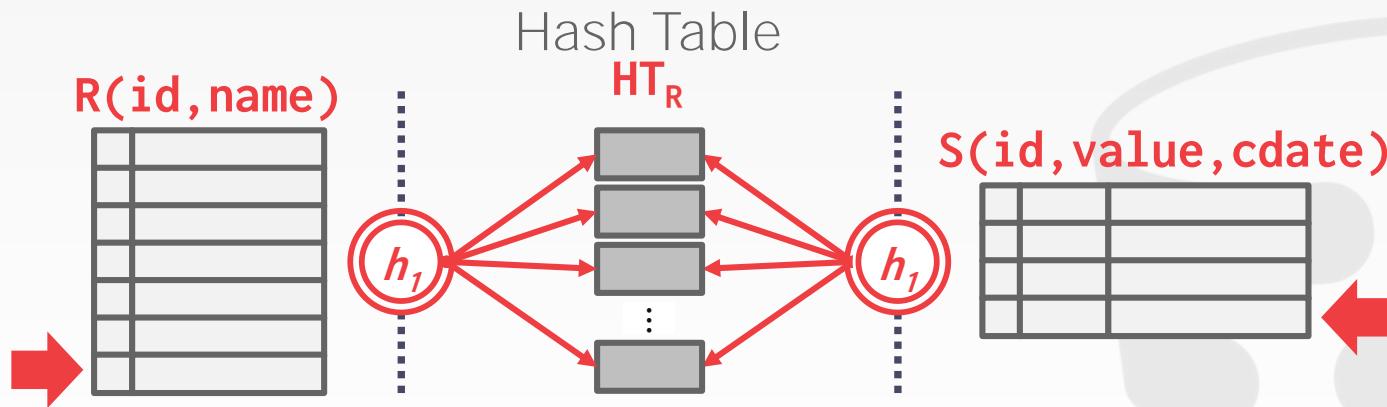
BASIC HASH JOIN ALGORITHM

```
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
    output, if  $h_1(s) \in HT_R$ 
```



BASIC HASH JOIN ALGORITHM

```
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
    output, if  $h_1(s) \in HT_R$ 
```



HASH TABLE CONTENTS

Key: The attribute(s) that the query is joining the tables on.

Value: Varies per implementation.

- Depends on what the operators above the join in the query plan expect as its input.



HASH TABLE VALUES

Approach #1: Full Tuple

- Avoid having to retrieve the outer relation's tuple contents on a match.
- Takes up more space in memory.

Approach #2: Tuple Identifier

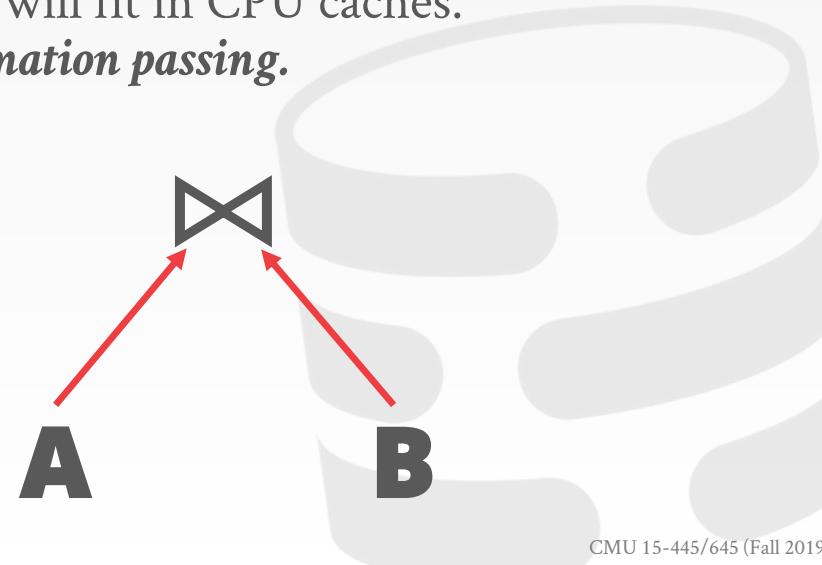
- Ideal for column stores because the DBMS doesn't fetch data from disk it doesn't need.
- Also better if join selectivity is low.



PROBE PHASE OPTIMIZATION

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

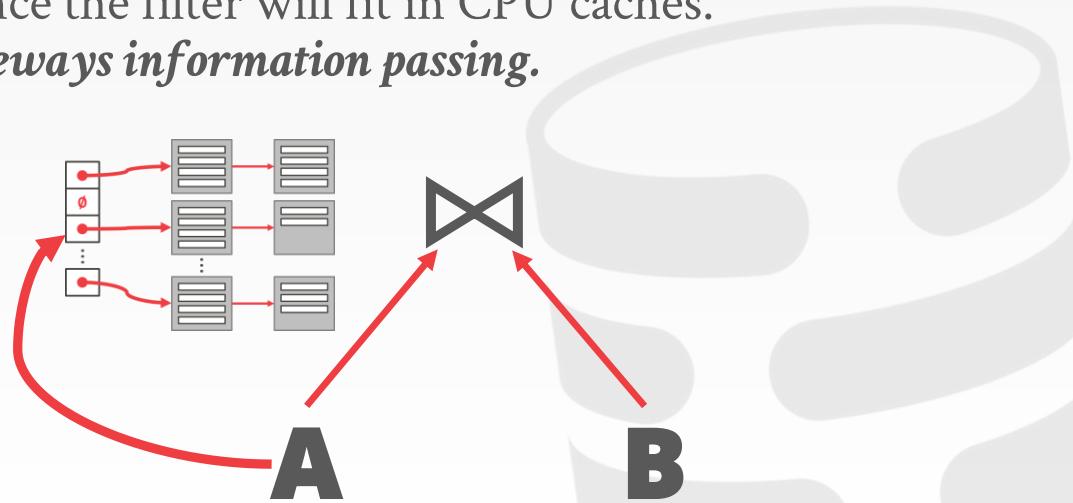
- Threads check the filter before probing the hash table.
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



PROBE PHASE OPTIMIZATION

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

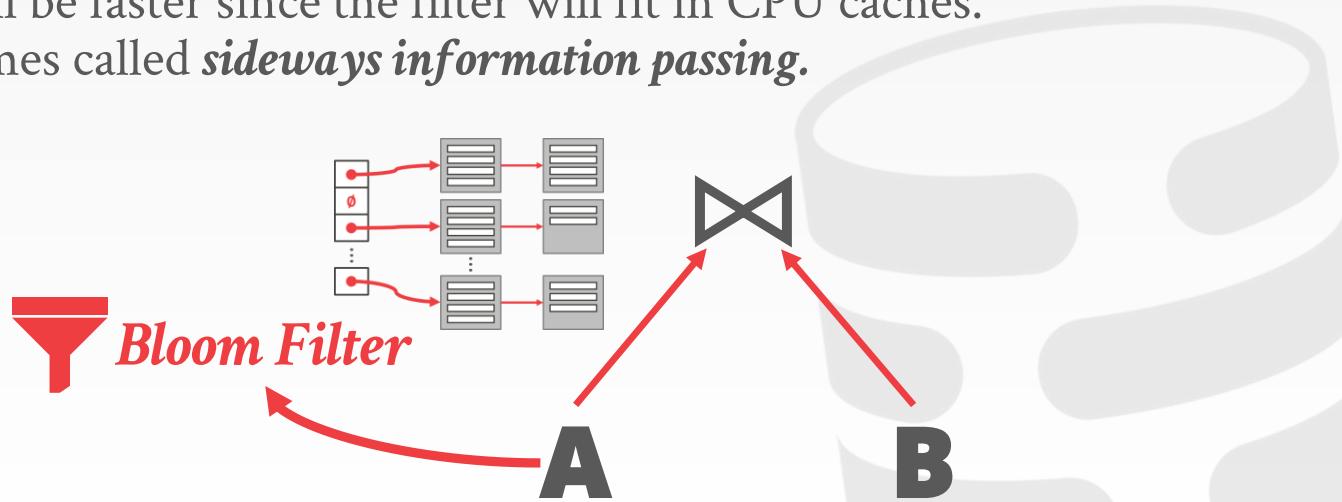
- Threads check the filter before probing the hash table.
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



PROBE PHASE OPTIMIZATION

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

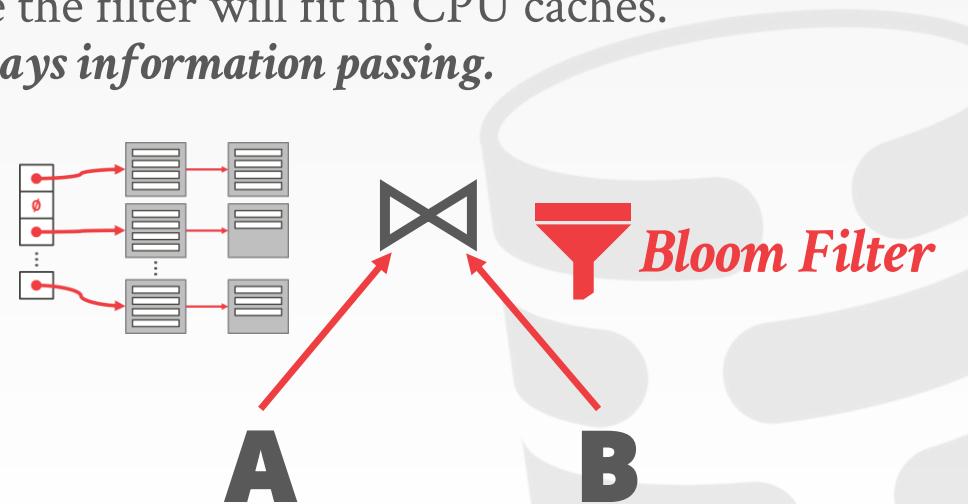
- Threads check the filter before probing the hash table.
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



PROBE PHASE OPTIMIZATION

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

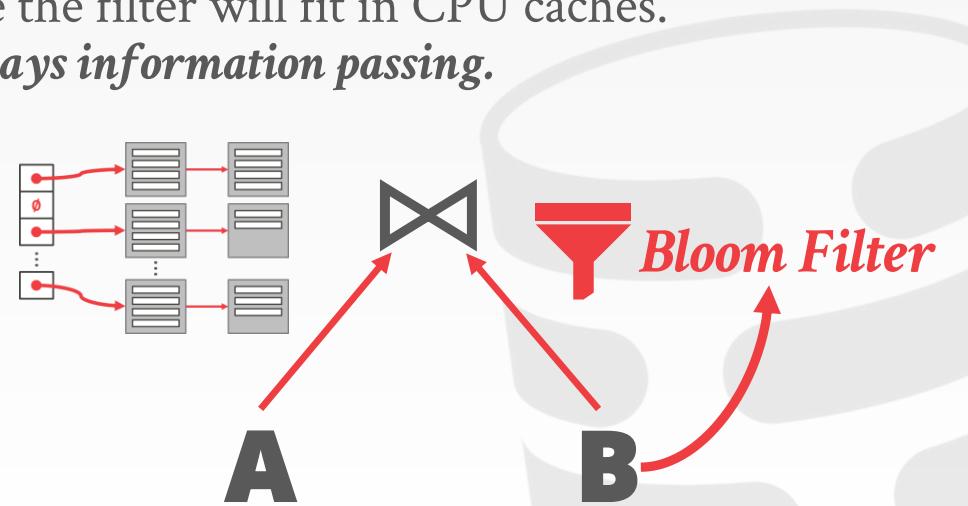
- Threads check the filter before probing the hash table.
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



PROBE PHASE OPTIMIZATION

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

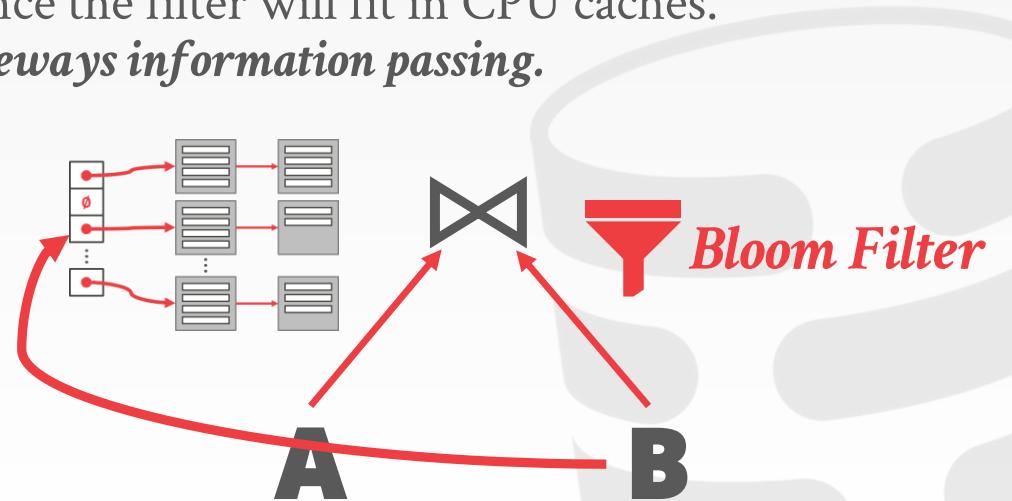
- Threads check the filter before probing the hash table.
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



PROBE PHASE OPTIMIZATION

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

- Threads check the filter before probing the hash table.
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



HASH JOIN

What happens if we do not have enough memory to fit the entire hash table?

We do not want to let the buffer pool manager swap out the hash table pages at a random.

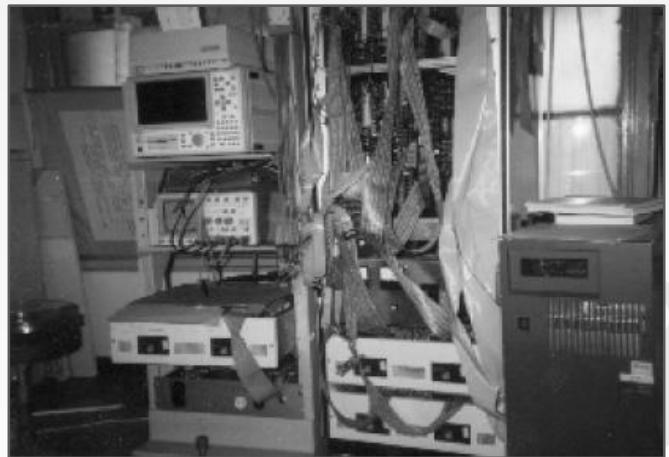


GRACE HASH JOIN

Hash join when tables do not fit in memory.

- **Build Phase:** Hash both tables on the join attribute into partitions.
- **Probe Phase:** Compares tuples in corresponding partitions for each table.

Named after the GRACE database machine from Japan in the 1980s.



GRACE
University of Tokyo

IBM DB2 Analytics Accelerator - GSE Management Summit

Choosing the best fit

Key indicators

IBM Netezza

- Performance and Price/performance leader
- Speed and ease of deployment and administration

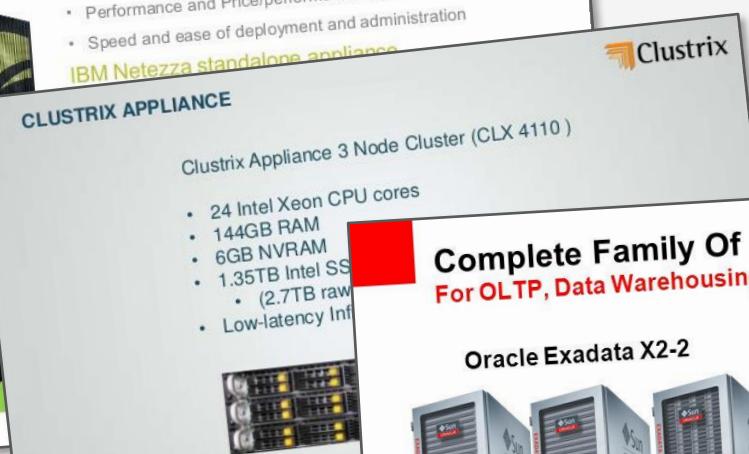
IBM Netezza standalone appliance



CLUSTRIX APPLIANCE

Clustrix Appliance 3 Node Cluster (CLX 4110)

- 24 Intel Xeon CPU cores
- 144GB RAM
- 6GB NVRAM
- 1.35TB Intel SSD
 - (2.7TB raw)
- Low-latency InfiniBand



HASH JOIN

Named after the GR
machine from Japan

Complete Family Of Database Machines
For OLTP, Data Warehousing & Consolidated Workloads



- Quarter, Half, Full and Multi-Racks
- Full and Multi-Racks

ORACLE



GRACE
University of Tokyo

IBM DB2 Analytics Accelerator - GSE Management Summit

Choosing the best fit

Key indicators

- IBM Netezza
- Performance and Price/performance leader
- Speed and ease of deployment and administration

IBM Netezza standalone appliance

CLUSTRIX APPLIANCE

Clustrix Appliance 3 Node Cluster (CLX 4110)

- 24 Intel Xeon CPU cores
- 144GB RAM
- 6GB NVRAM
- 1.35TB Intel SSD
 - (2.7TB raw)
- Low-latency InfiniBand

Complete Family Of
For OLTP, Data Warehousing

Named after the GR
machine from Japan

HASH JOIN

Yellowbrick Data Warehouse Architecture

Real-time Feeds
Ingest IoT or OLTP data
Capture 100,000s of rows per second

Periodic Bulk Loads
Capture terabytes of data, petabytes over time

Load and Transform
Use existing ETL tools including intensive push-down ETL

Interactive Applications
Serve short queries in under 100 milliseconds

Powerful Analytics
Respond to complex BI queries in just a few seconds

Business Critical Reporting
Workload management for prioritized responses

Source: yellowbrickdata.com

GRACE
University of Tokyo

ORACLE

Oracle Exadata X2-2

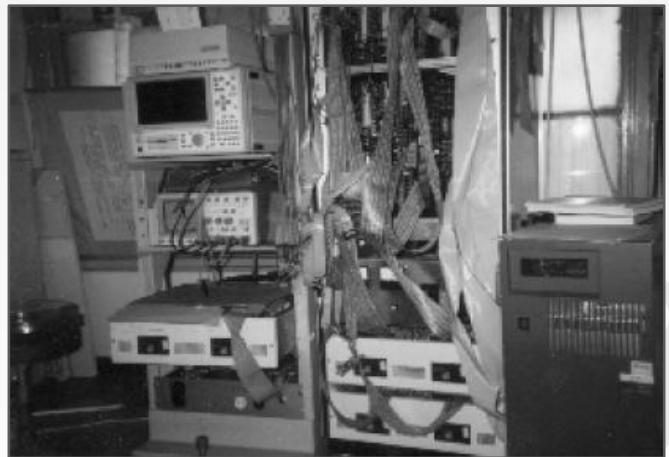
- Quarter, Half, Full and Multi-Racks
- Full and Multi-Racks

GRACE HASH JOIN

Hash join when tables do not fit in memory.

- **Build Phase:** Hash both tables on the join attribute into partitions.
- **Probe Phase:** Compares tuples in corresponding partitions for each table.

Named after the GRACE database machine from Japan in the 1980s.

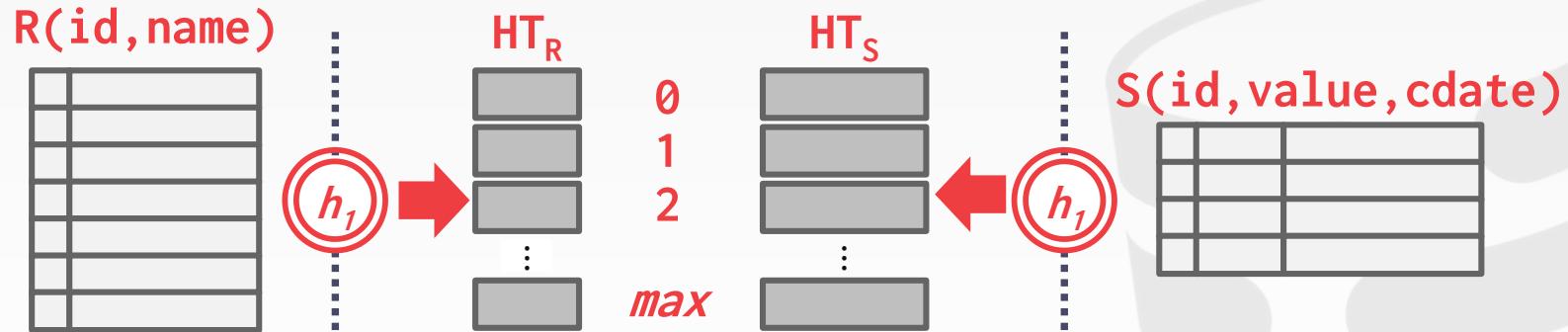


GRACE
University of Tokyo

GRACE HASH JOIN

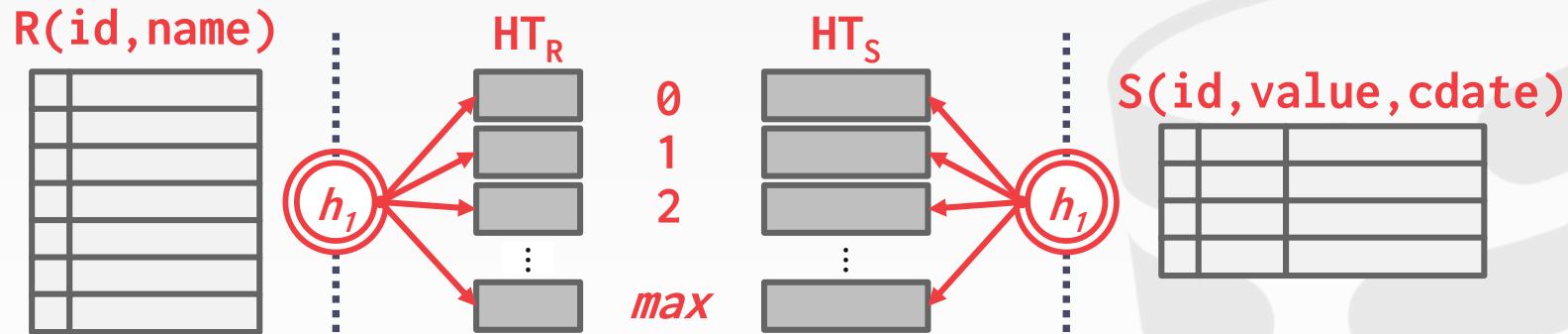
Hash **R** into $(0, 1, \dots, max)$ buckets.

Hash **S** into the same # of buckets with the same hash function.



GRACE HASH JOIN

Join each pair of matching buckets
between **R** and **S**.



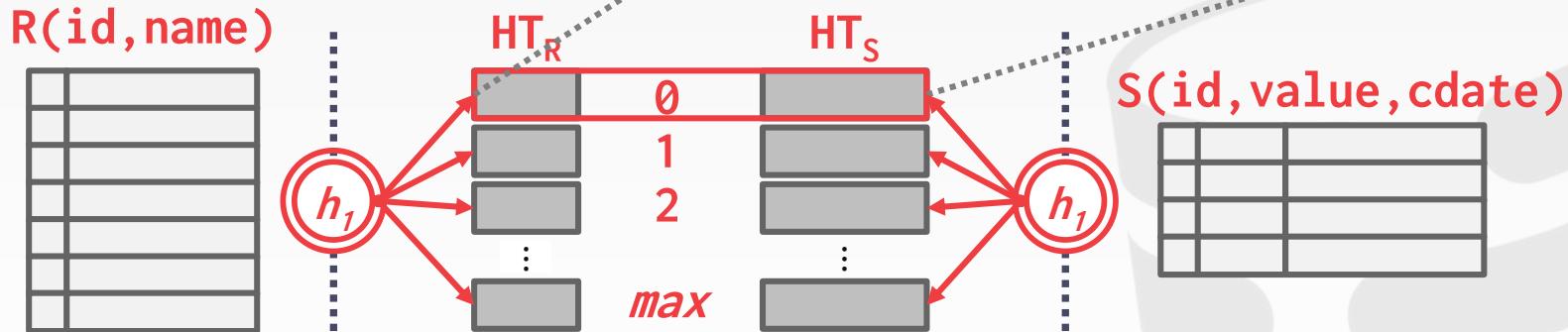
GRACE HASH JOIN

Join each pair of matching buckets between **R** and **S**.

```
foreach tuple  $r \in \text{bucket}_{R,0}$ :  

foreach tuple  $s \in \text{bucket}_{S,0}$ :  

  emit, if  $\text{match}(r, s)$ 
```



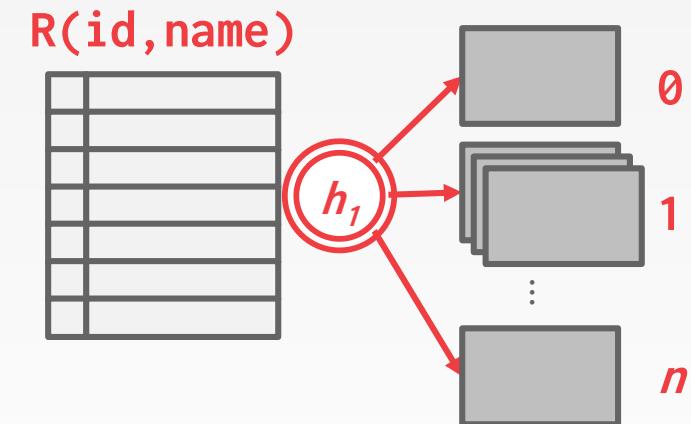
GRACE HASH JOIN

If the buckets do not fit in memory, then use **recursive partitioning** to split the tables into chunks that will fit.

- Build another hash table for **$\text{bucket}_{R,i}$** using hash function **h_2** (with **$h_2 \neq h_1$**).
- Then probe it for each tuple of the other table's bucket at that level.

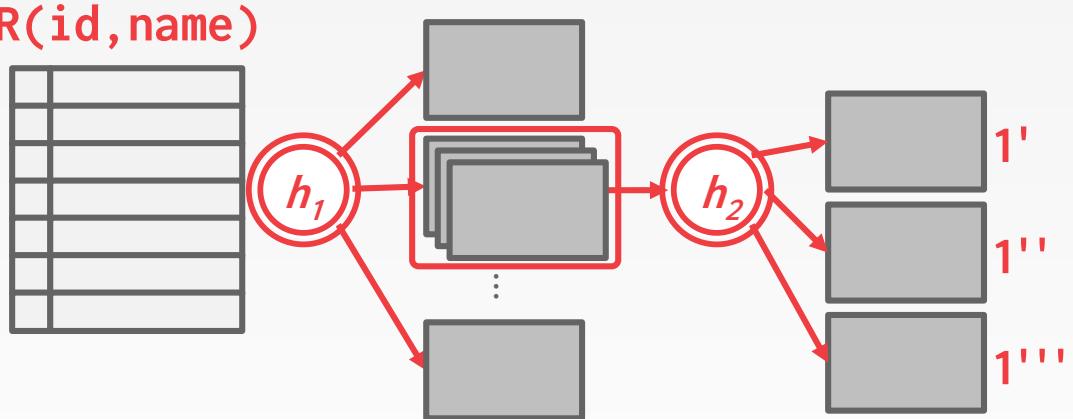


RECURSIVE PARTITIONING

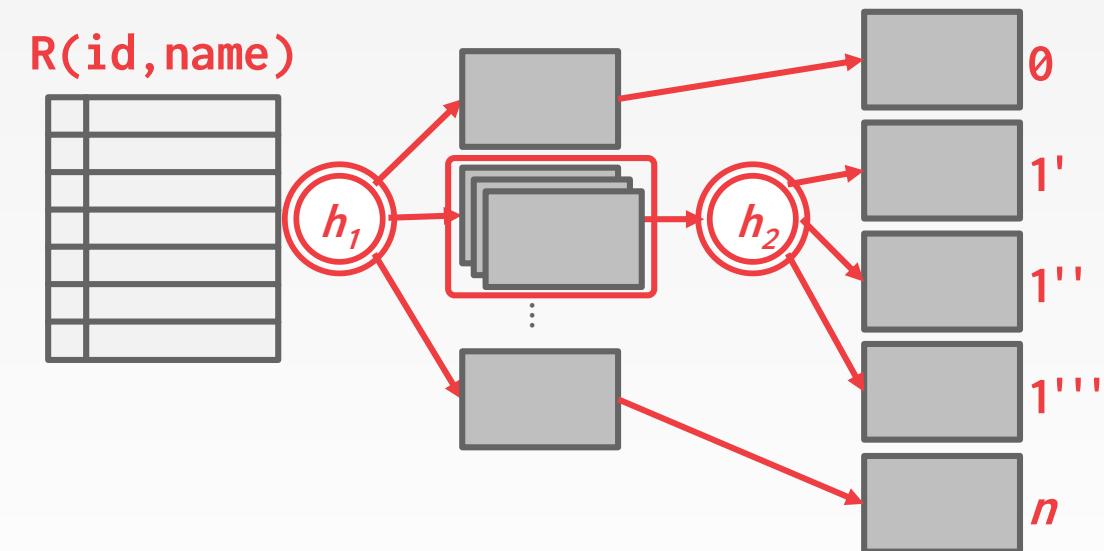


RECURSIVE PARTITIONING

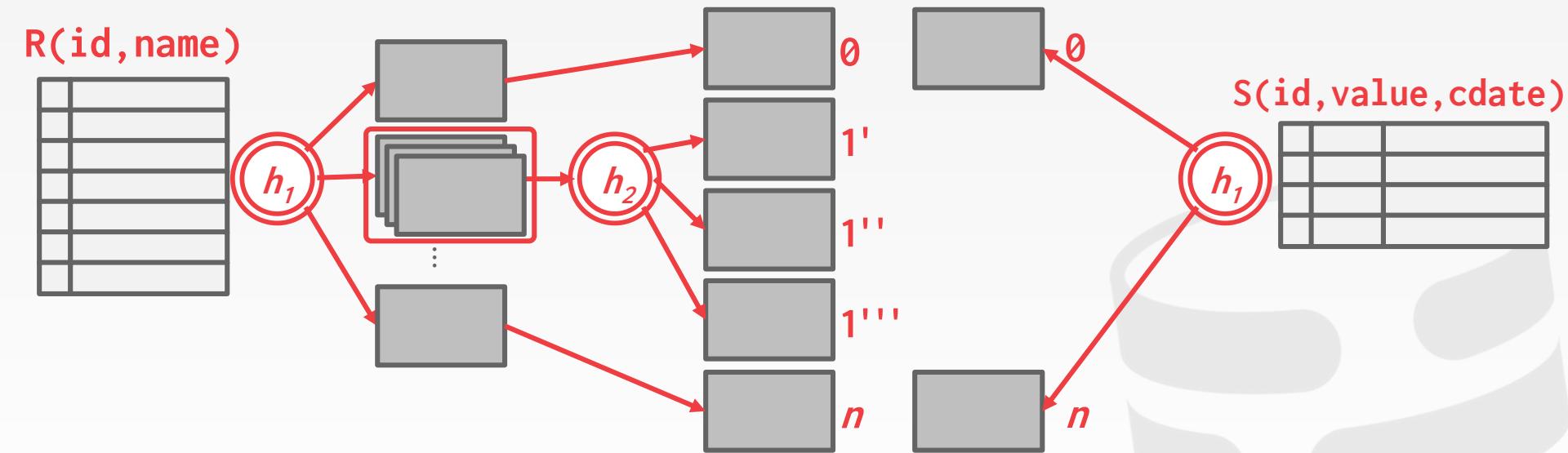
$R(id, name)$



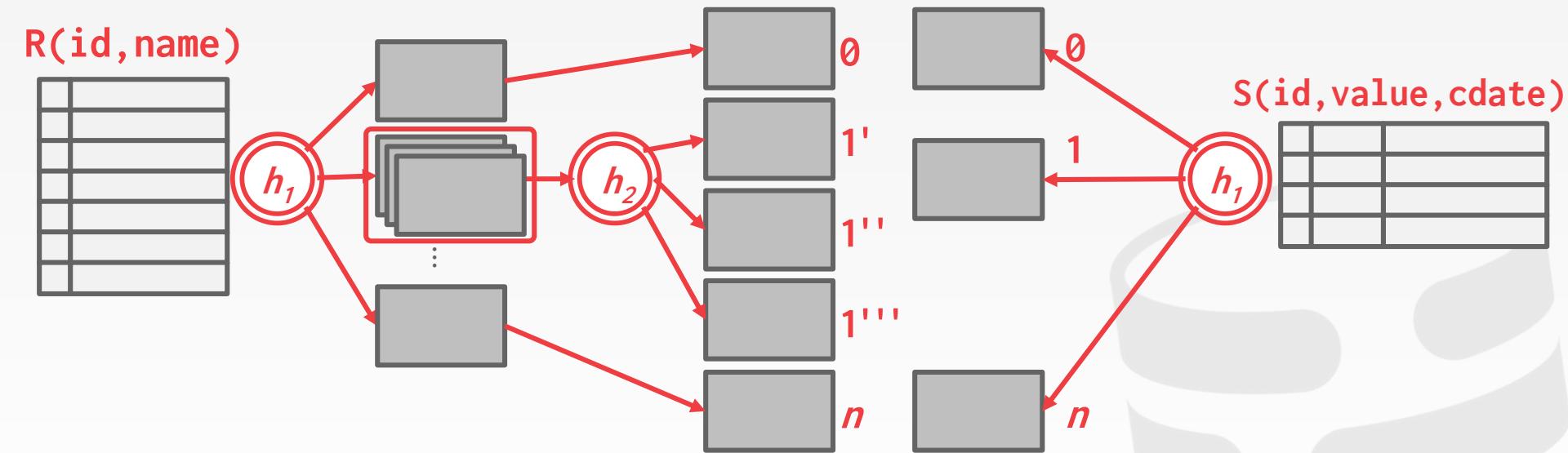
RECURSIVE PARTITIONING



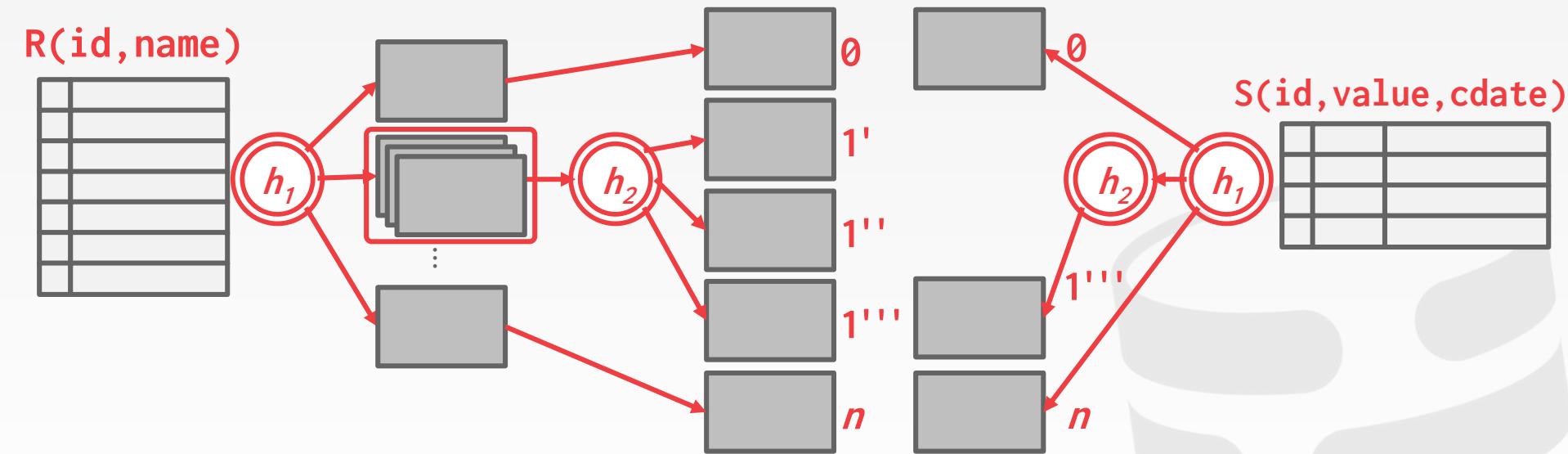
RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



GRACE HASH JOIN

Cost of hash join?

- Assume that we have enough buffers.
- Cost: $3(M + N)$

Partitioning Phase:

- Read+Write both tables
- $2(M+N)$ IOs

Probing Phase:

- Read both tables
- $M+N$ IOs



GRACE HASH JOIN

Example database:

- $M = 1000$, $m = 100,000$
- $N = 500$, $n = 40,000$

Cost Analysis:

- $3 \cdot (M + N) = 3 \cdot (1000 + 500) = 4,500 \text{ IOs}$
- At 0.1 ms/IO, Total time ≈ 0.45 seconds



OBSERVATION

If the DBMS knows the size of the outer table,
then it can use a static hash table.

→ Less computational overhead for build / probe
operations.

If we do not know the size, then we have to use a
dynamic hash table or allow for overflow pages.

JOIN ALGORITHMS: SUMMARY

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (M \cdot \log N)$	20 seconds
Sort-Merge Join	$M + N + (\text{sort cost})$	0.59 seconds
Hash Join	$3(M + N)$	0.45 seconds

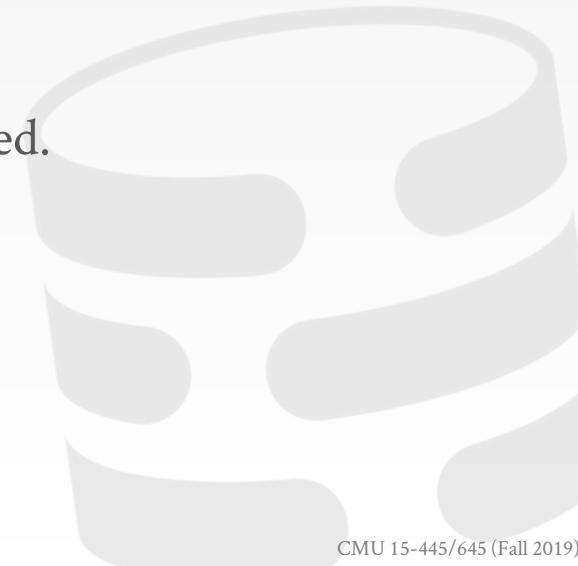
CONCLUSION

Hashing is almost always better than sorting for operator execution.

Caveats:

- Sorting is better on non-uniform data.
- Sorting is better when result needs to be sorted.

Good DBMSs use either or both.



NEXT CLASS

Composing operators together to execute queries.



12

Query Execution – Part I



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #3 is due Wed Oct 9th @ 11:59pm

Mid-Term Exam is Wed Oct 16th @ 12:00pm

Project #2 is due Sun Oct 20th @ 11:59pm



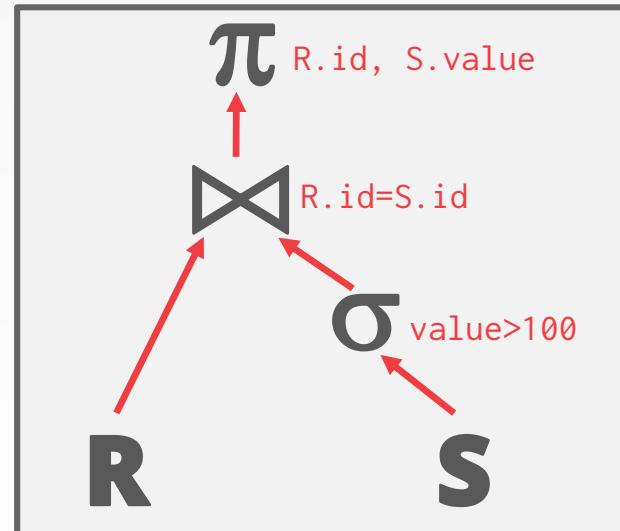
QUERY PLAN

The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

The output of the root node is the result of the query.

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



TODAY'S AGENDA

Processing Models
Access Methods
Expression Evaluation



PROCESSING MODEL

A DBMS's **processing model** defines how the system executes a query plan.
→ Different trade-offs for different workloads.

Approach #1: Iterator Model

Approach #2: Materialization Model

Approach #3: Vectorized / Batch Model



ITERATOR MODEL

Each query plan operator implements a **Next** function.

- On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples.
- The operator implements a loop that calls next on its children to retrieve their tuples and then process them.

Also called **Volcano** or **Pipeline** Model.

ITERATOR MODEL

Next()

```
for t in child.Next():
    emit(projection(t))
```

Next()

```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1◁◁ t2)
```

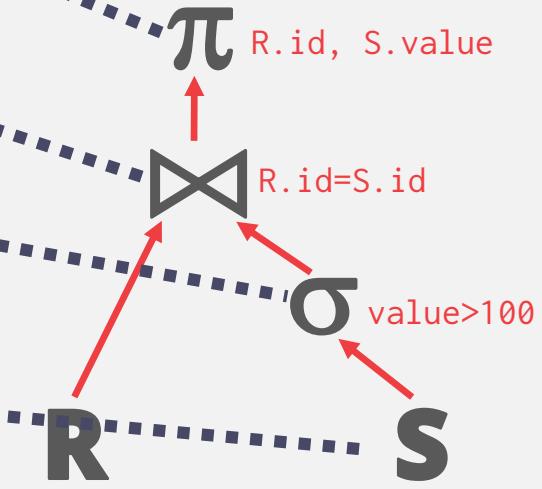
Next()

```
for t in R:
    emit(t)
```

Next()

```
for t in S:
    emit(t)
```

**SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100**



ITERATOR MODEL

1

```
for t in child.Next():
    emit(projection(t))
```

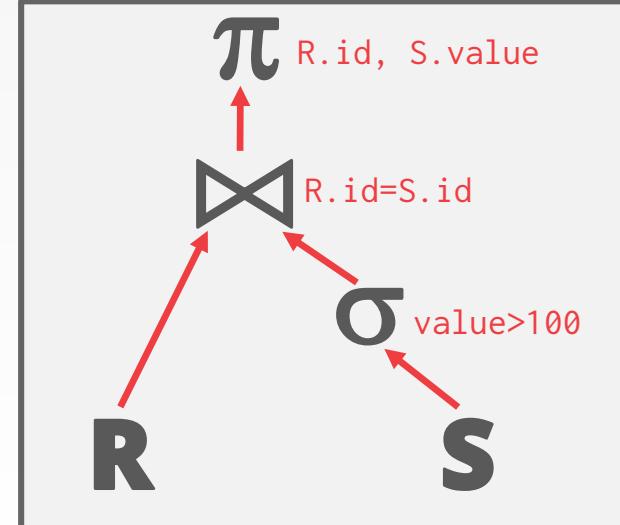
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1◁◁ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



ITERATOR MODEL

1

```
for t in child.Next():
    emit(projection(t))
```

2

```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1◁ t2)
```

```
for t in child.Next():
    if evalPred(t): emit(t)
```

```
for t in R:
    emit(t)
```

```
for t in S:
    emit(t)
```

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

$\pi_{R.id, S.value}$

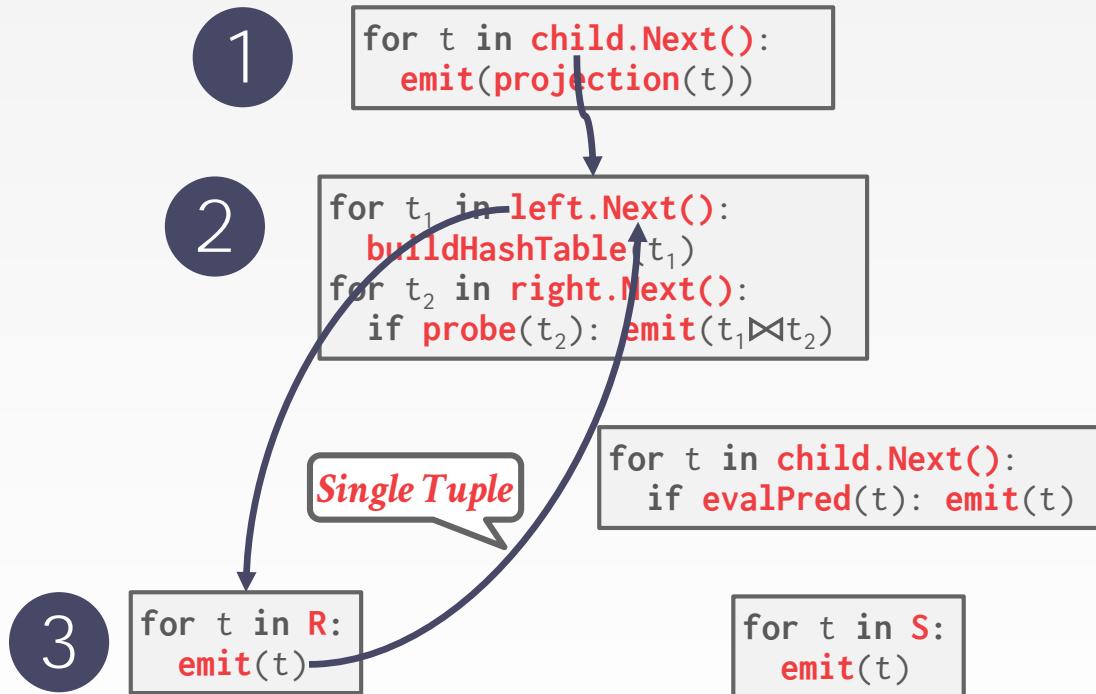
$\bowtie_{R.id=S.id}$

$\sigma_{value>100}$

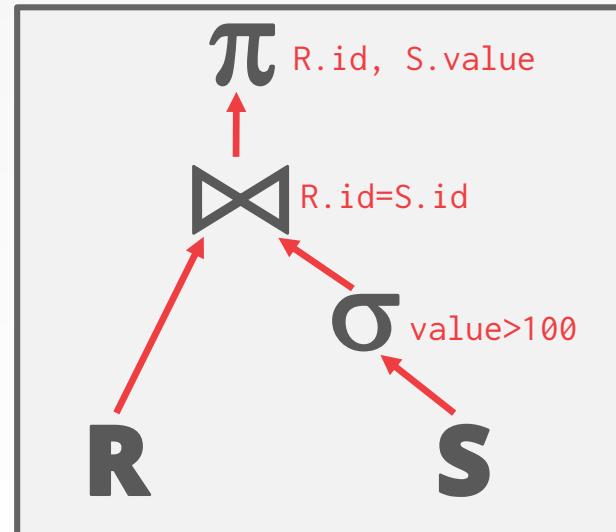
R

S

ITERATOR MODEL



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



ITERATOR MODEL

1

```
for t in child.Next():
    emit(projection(t))
```

2

```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1◁ t2)
```

3

```
for t in R:
    emit(t)
```

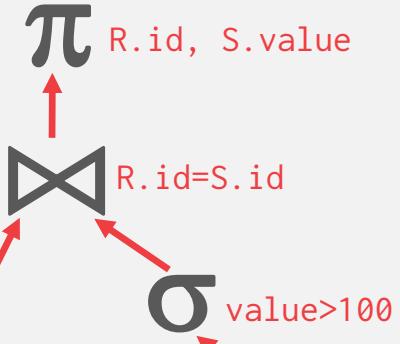
```
for t in child.Next():
    if evalPred(t): emit(t)
```

5

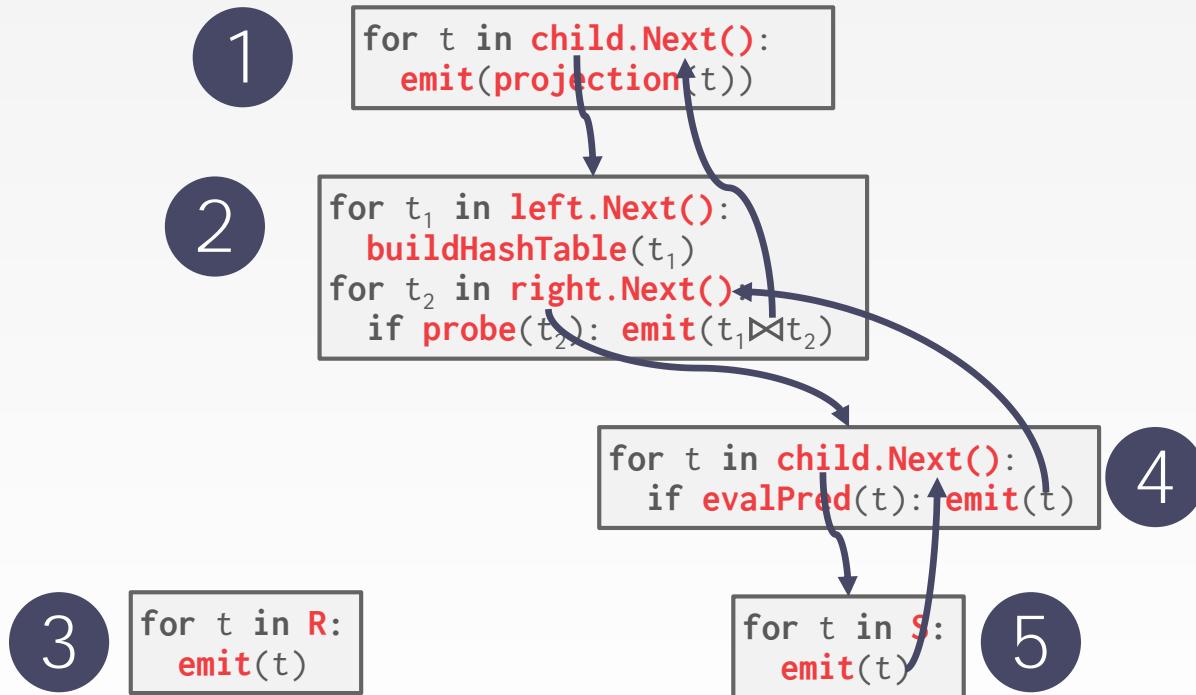
```
for t in S:
    emit(t)
```

**SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100**

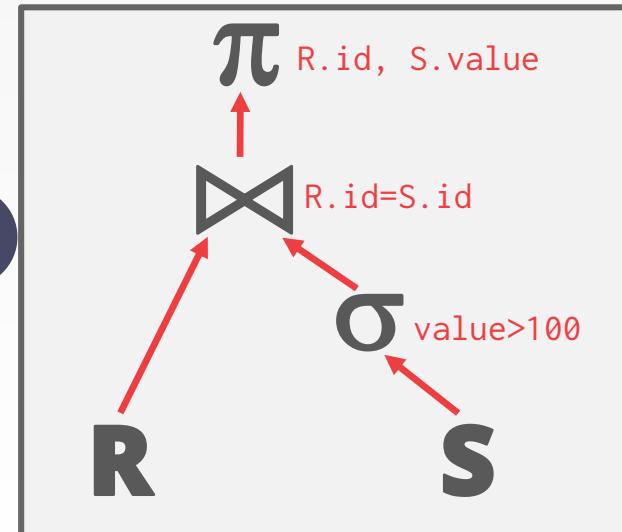
4



ITERATOR MODEL



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



ITERATOR MODEL

This is used in almost every DBMS. Allows for tuple **pipelining**.

Some operators have to block until their children emit all of their tuples.
→ Joins, Subqueries, Order By

Output control works easily with this approach.



MATERIALIZATION MODEL

Each operator processes its input all at once and then emits its output all at once.

- The operator "materializes" its output as a single result.
- The DBMS can push down hints into to avoid scanning too many tuples.
- Can send either a materialized row or a single column.

The output can be either whole tuples (NSM) or subsets of columns (DSM)

MATERIALIZATION MODEL

1

```
out = []
for t in child.Output():
    out.add(projection(t))
return out
```

```
out = []
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1◁◁ t2)
return out
```

```
out = []
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = []
for t in R:
    out.add(t)
return out
```

```
out = []
for t in S:
    out.add(t)
return out
```

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

π R.id, S.value

\bowtie R.id=S.id

σ value>100

R

S

MATERIALIZATION MODEL

1

```
out = []
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = []
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1◁ t2)
return out
```

3

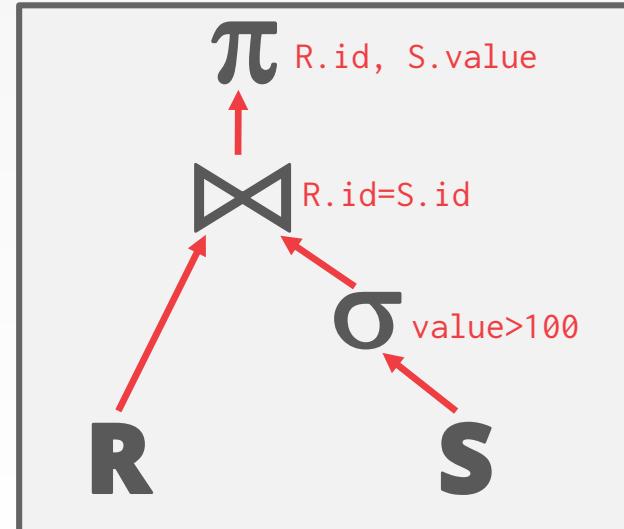
```
out = []
for t in R:
    out.add(t)
return out
```

All Tuples

```
out = []
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = []
for t in S:
    out.add(t)
return out
```

SELECT R.id, S.cdate
FROM R **JOIN** S
ON R.id = S.id
WHERE S.value > 100



MATERIALIZATION MODEL

1

```
out = []
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = []
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1◁ t2)
return out
```

3

```
out = []
for t in R:
    out.add(t)
return out
```

4

```
out = []
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

5

```
out = []
for t in S:
    out.add(t)
return out
```

SELECT R.id, S.cdate
FROM R **JOIN** S
ON R.id = S.id
WHERE S.value > 100

π R.id, S.value

\bowtie R.id=S.id

σ value>100

R

S

MATERIALIZATION MODEL

1

```
out = []
for t in child.Output():
    out.add(projection(t))
return out
```

2

```
out = []
for t1 in left.Output():
    buildHashTable(t1)
for t2 in right.Output():
    if probe(t2): out.add(t1◁ t2)
return out
```

3

```
out = []
for t in R:
    out.add(t)
return out
```

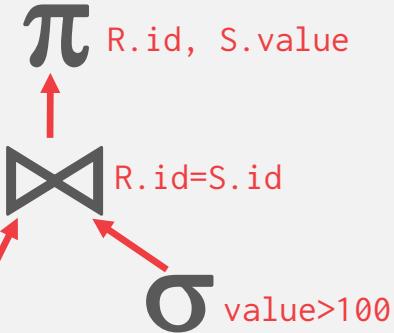
**SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100**

4

```
out = []
for t in child.Output():
    if evalPred(t): out.add(t)
return out
```

```
out = []
for t in S:
    out.add(t)
return out
```

5



MATERIALIZATION MODEL

Better for OLTP workloads because queries only access a small number of tuples at a time.

- Lower execution / coordination overhead.
- Fewer function calls.

Not good for OLAP queries with large intermediate results.



VECTORIZATION MODEL

Like the Iterator Model where each operator implements a **Next** function in this model.

Each operator emits a batch of tuples instead of a single tuple.

- The operator's internal loop processes multiple tuples at a time.
- The size of the batch can vary based on hardware or query properties.

VECTORIZATION MODEL

```
out = []
for t in child.Next():
    out.add(projection(t))
    if |out|>n: emit(out)
```

1

```
out = []
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): out.add(t1◁ t2)
    if |out|>n: emit(out)
```

2

```
out = []
for t in child.Next():
    if evalPred(t): out.add(t)
    if |out|>n: emit(out)
```

```
out = []
for t in S:
    out.add(t)
    if |out|>n: emit(out)
```

3

Tuple Batch

**SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100**

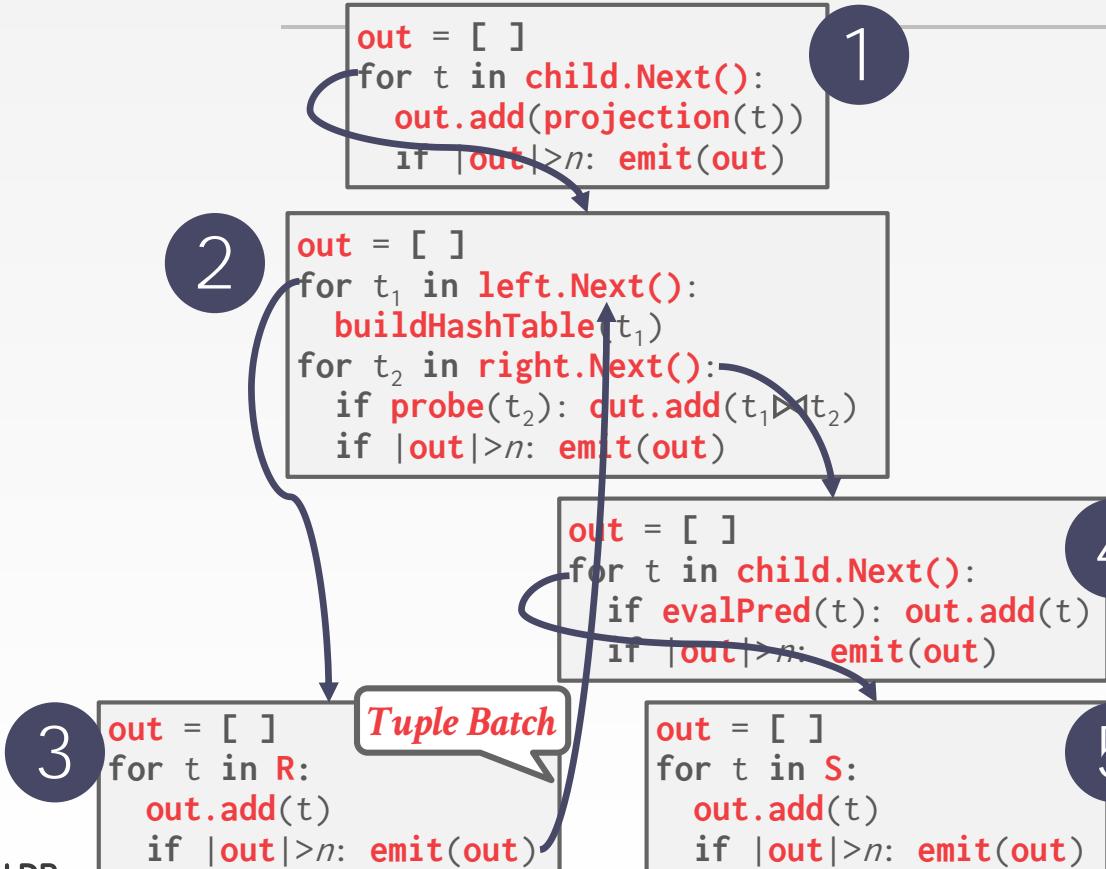
π R.id, S.value

\bowtie R.id=S.id

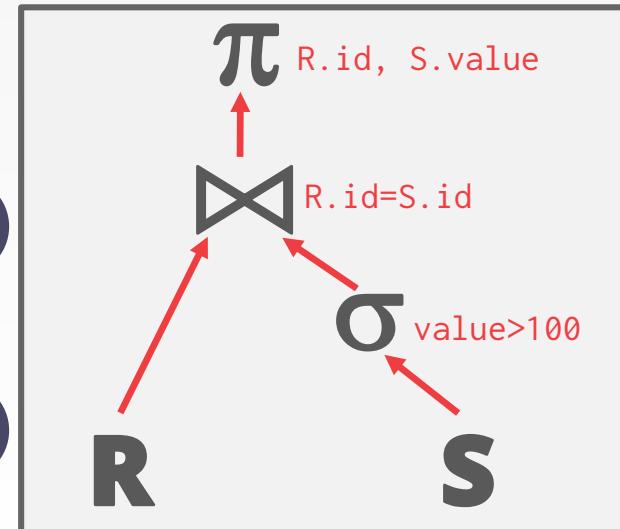
σ value>100

R**S**

VECTORIZATION MODEL



SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100



VECTORIZATION MODEL

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.



PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom

- Start with the root and "pull" data up from its children.
- Tuples are always passed with function calls.

Approach #2: Bottom-to-Top

- Start with leaf nodes and push data to their parents.
- Allows for tighter control of caches/registers in pipelines.

ACCESS METHODS

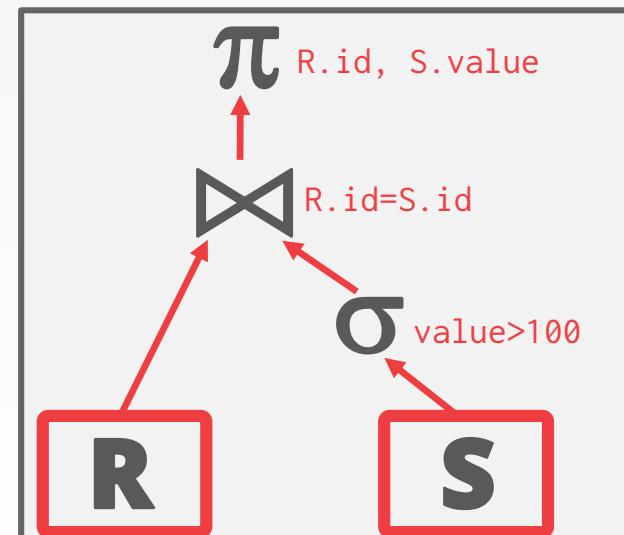
An **access method** is a way that the DBMS can access the data stored in a table.

→ Not defined in relational algebra.

Three basic approaches:

- Sequential Scan
- Index Scan
- Multi-Index / "Bitmap" Scan

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



SEQUENTIAL SCAN

For each page in the table:

- Retrieve it from the buffer pool.
- Iterate over each tuple and check whether to include it.

The DBMS maintains an internal **cursor** that tracks the last page / slot it examined.

```
for page in table.pages:  
    for t in page.tuples:  
        if evalPred(t):  
            // Do Something!
```

SEQUENTIAL SCAN: OPTIMIZATIONS

This is almost always the worst thing that the DBMS can do to execute a query.

Sequential Scan Optimizations:

- Prefetching
- Buffer Pool Bypass
- Parallelization
- Zone Maps
- Late Materialization
- Heap Clustering





ZONE MAPS



```
SELECT * FROM table  
WHERE val > 600
```

Pre-computed aggregates for the attribute values in a page. DBMS checks the zone map first to decide whether it wants to access the page.

Original Data

val
100
200
300
400
400



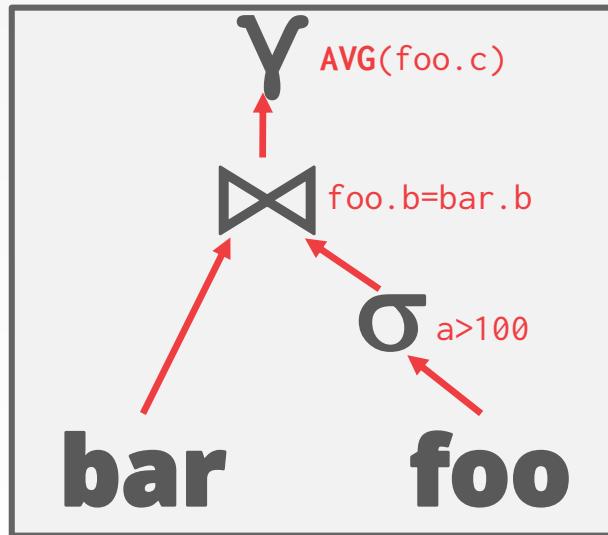
Zone Map

type	val
MIN	100
MAX	400
AVG	280
SUM	1400
COUNT	5



LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

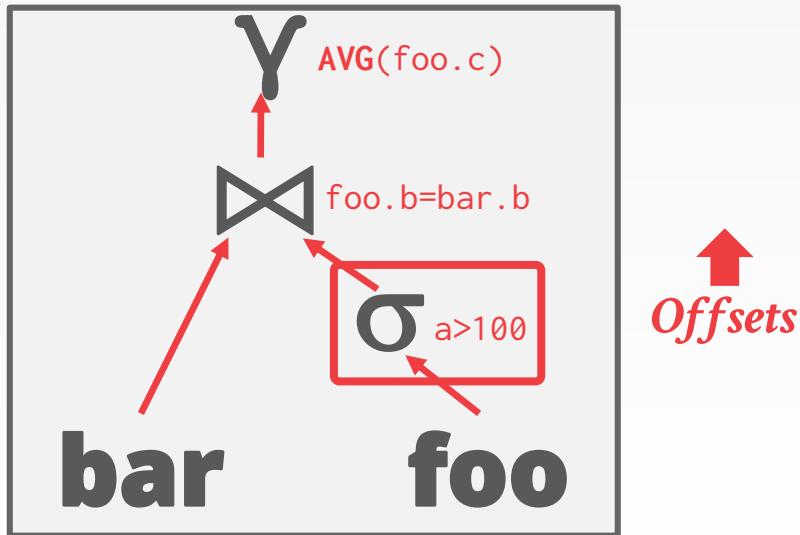


```
SELECT AVG(foo.c)
FROM foo JOIN bar
ON foo.b = bar.b
WHERE foo.a > 100
```

	a	b	c
0			
1			
2			
3			

LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

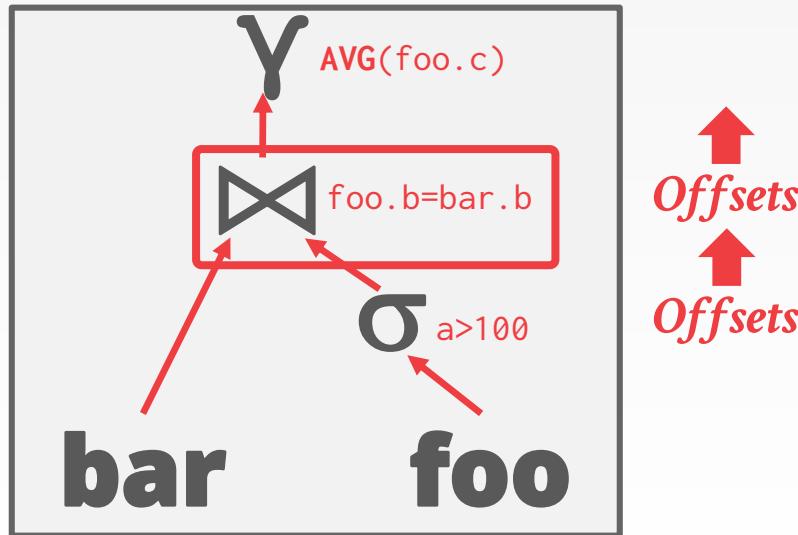


```
SELECT AVG(foo.c)
FROM foo JOIN bar
ON foo.b = bar.b
WHERE foo.a > 100
```

	a	b	c
0			
1			
2			
3			

LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.

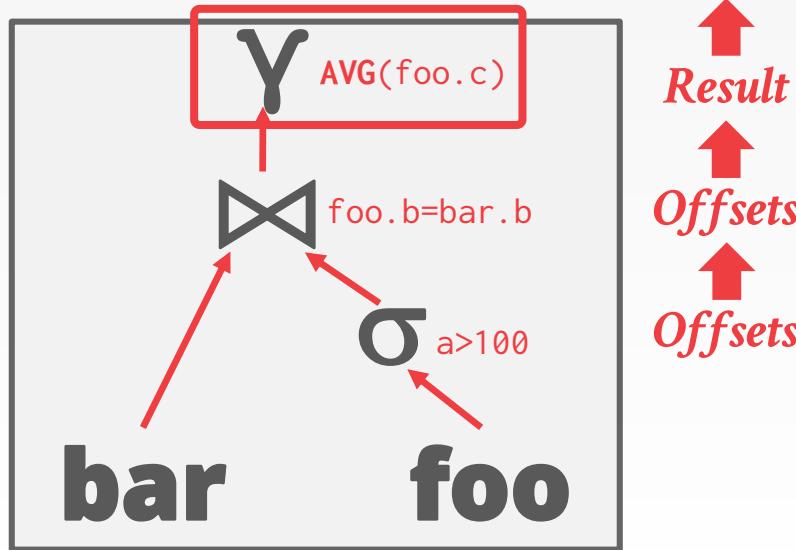


```
SELECT AVG(foo.c)
FROM foo JOIN bar
ON foo.b = bar.b
WHERE foo.a > 100
```

A table with three columns labeled "a", "b", and "c". The first column has four rows labeled 0, 1, 2, and 3. The second column, "b", has four rows and is highlighted with a red border. The third column has four rows. The entire table is set against a grey background.

LATE MATERIALIZATION

DSM DBMSs can delay stitching together tuples until the upper parts of the query plan.



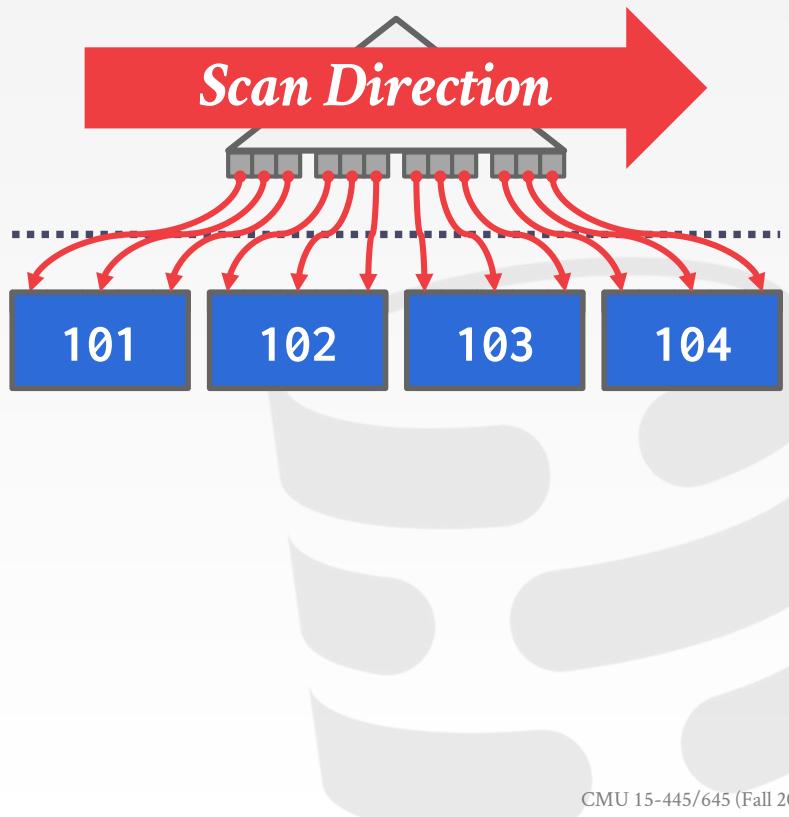
```
SELECT AVG(foo.c)
FROM foo JOIN bar
ON foo.b = bar.b
WHERE foo.a > 100
```

	a	b	c
0			
1			
2			
3			

HEAP CLUSTERING

Tuples are sorted in the heap's pages using the order specified by a clustering index.

If the query accesses tuples using the clustering index's attributes, then the DBMS can jump directly to the pages that it needs.



INDEX SCAN

The DBMS picks an index to find the tuples that the query needs.

Lecture 14

Which index to use depends on:

- What attributes the index contains
- What attributes the query references
- The attribute's value domains
- Predicate composition
- Whether the index has unique or non-unique keys

INDEX SCAN

Suppose that we have a single table with 100 tuples and two indexes:

- Index #1: **age**
- Index #2: **dept**

```
SELECT * FROM students  
WHERE age < 30  
    AND dept = 'CS'  
    AND country = 'US'
```

Scenario #1

There are 99 people under the age of 30 but only 2 people in the CS department.

Scenario #2

There are 99 people in the CS department but only 2 people under the age of 30.

MULTI-INDEX SCAN

If there are multiple indexes that the DBMS can use for a query:

- Compute sets of record ids using each matching index.
- Combine these sets based on the query's predicates (union vs. intersect).
- Retrieve the records and apply any remaining predicates.

Postgres calls this Bitmap Scan.

MULTI-INDEX SCAN

With an index on **age** and an index on **dept**,

- We can retrieve the record ids satisfying **age<30** using the first,
- Then retrieve the record ids satisfying **dept='CS'** using the second,
- Take their intersection
- Retrieve records and check **country='US'**.

```
SELECT * FROM students
WHERE age < 30
    AND dept = 'CS'
    AND country = 'US'
```



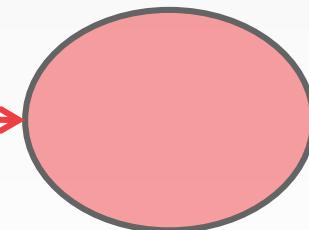
MULTI-INDEX SCAN

Set intersection can be done with bitmaps, hash tables, or Bloom filters.



age < 30

record ids



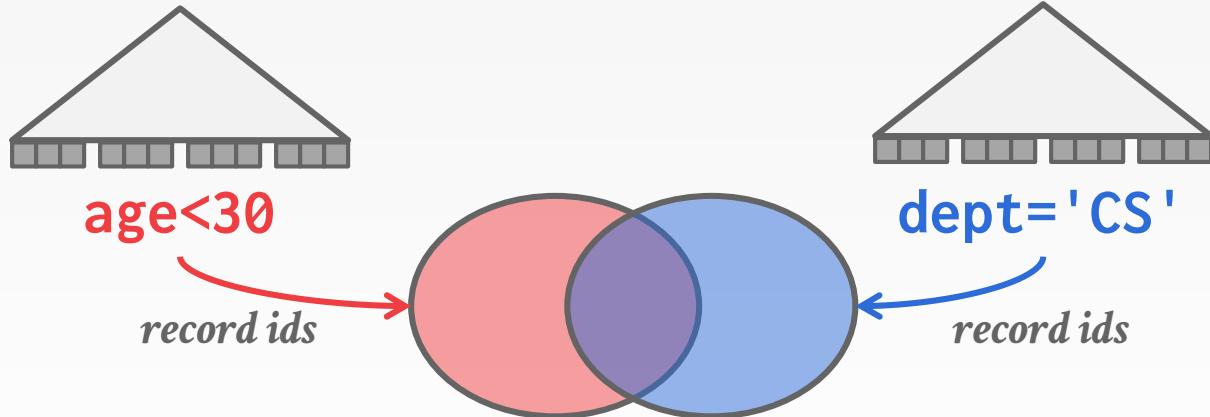
dept = 'CS'

```
SELECT * FROM students  
WHERE age < 30  
AND dept = 'CS'  
AND country = 'US'
```



MULTI-INDEX SCAN

Set intersection can be done with bitmaps, hash tables, or Bloom filters.

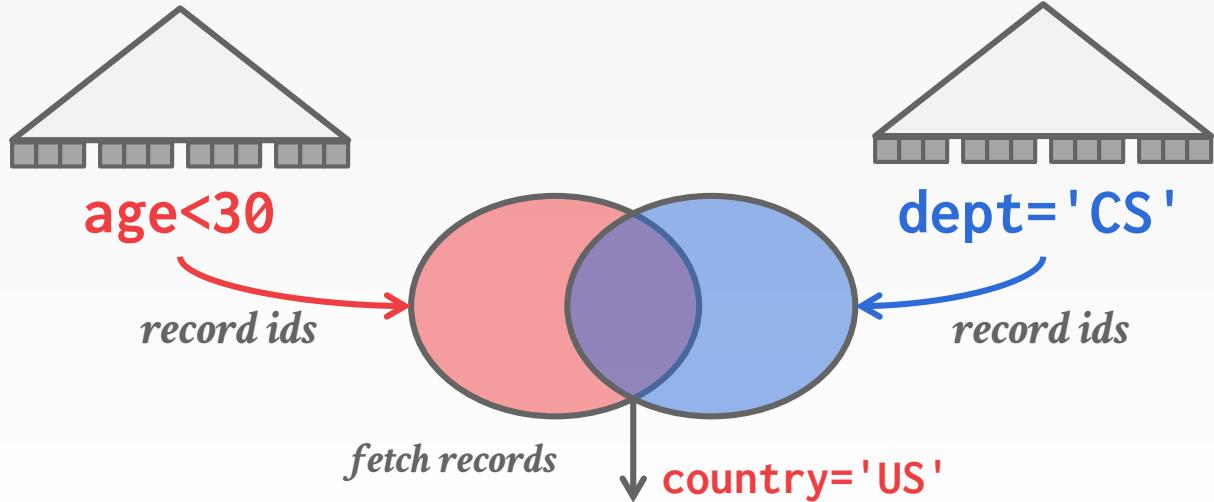


```
SELECT * FROM students  
WHERE age < 30  
AND dept = 'CS'  
AND country = 'US'
```



MULTI-INDEX SCAN

Set intersection can be done with bitmaps, hash tables, or Bloom filters.

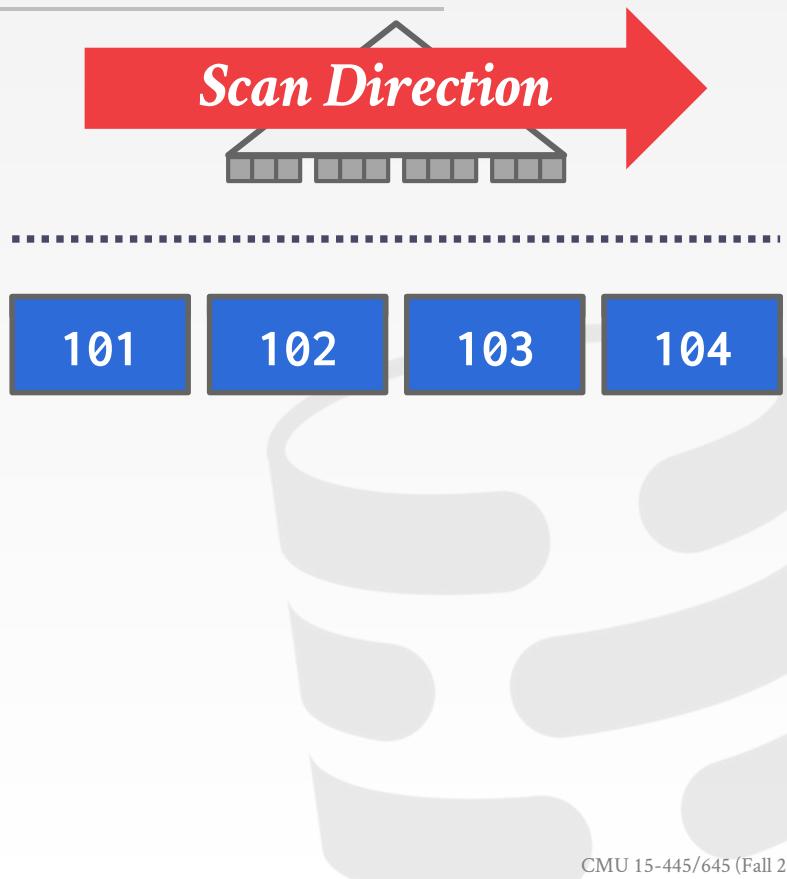


```
SELECT * FROM students  
WHERE age < 30  
AND dept = 'CS'  
AND country = 'US'
```

INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

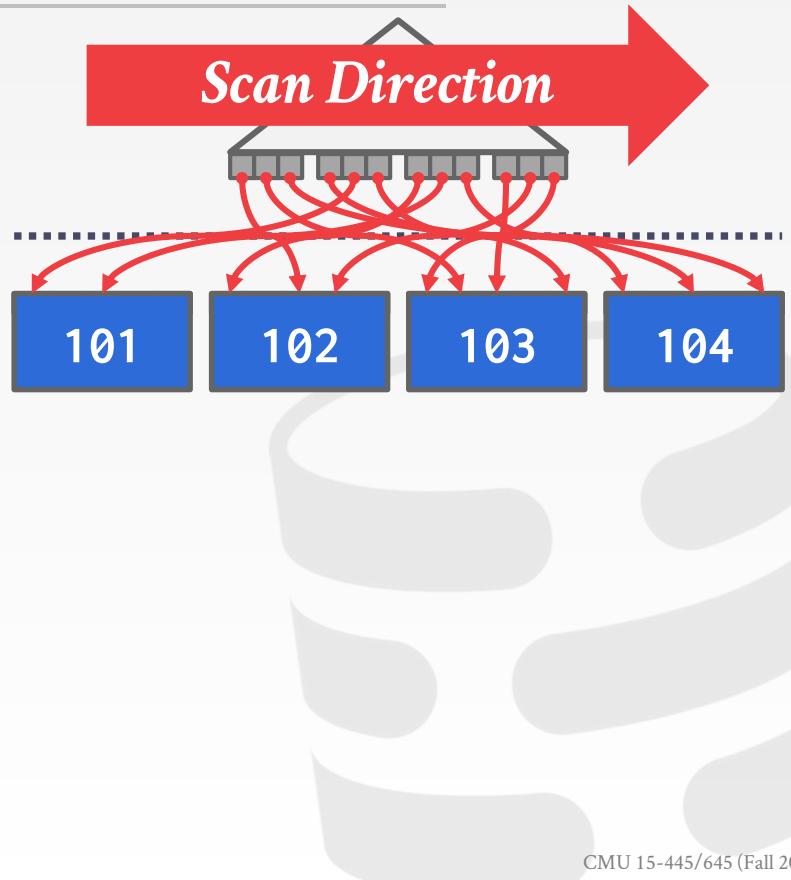
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

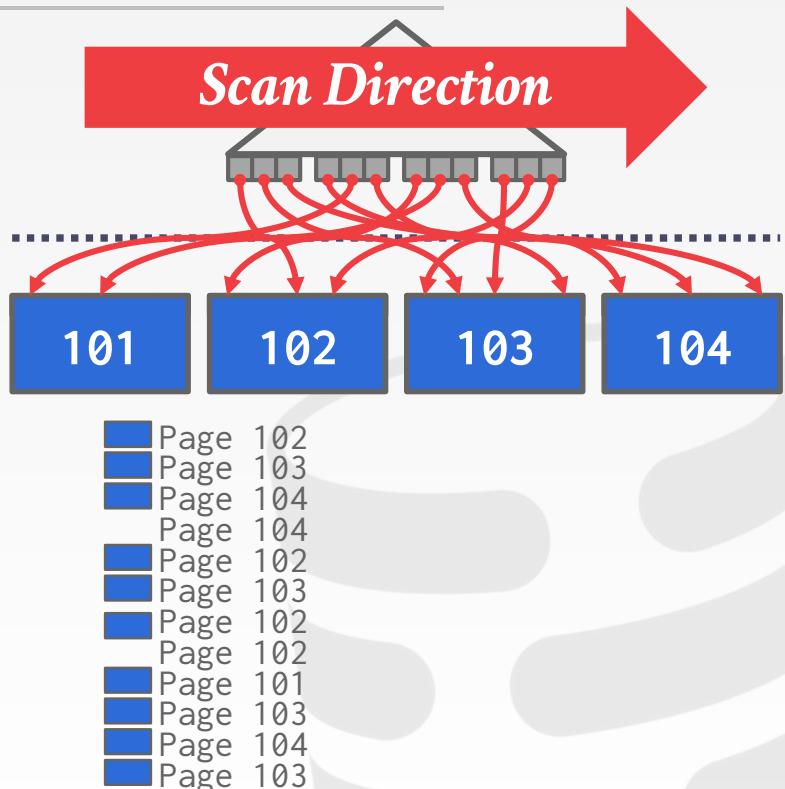
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

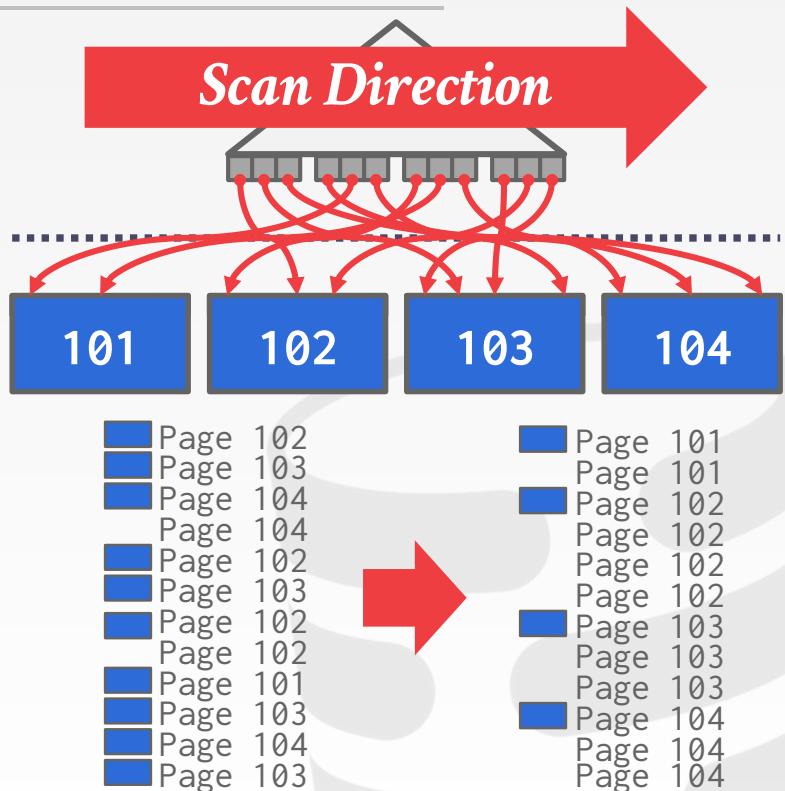
The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



INDEX SCAN PAGE SORTING

Retrieving tuples in the order that appear in an unclustered index is inefficient.

The DBMS can first figure out all the tuples that it needs and then sort them based on their page id.



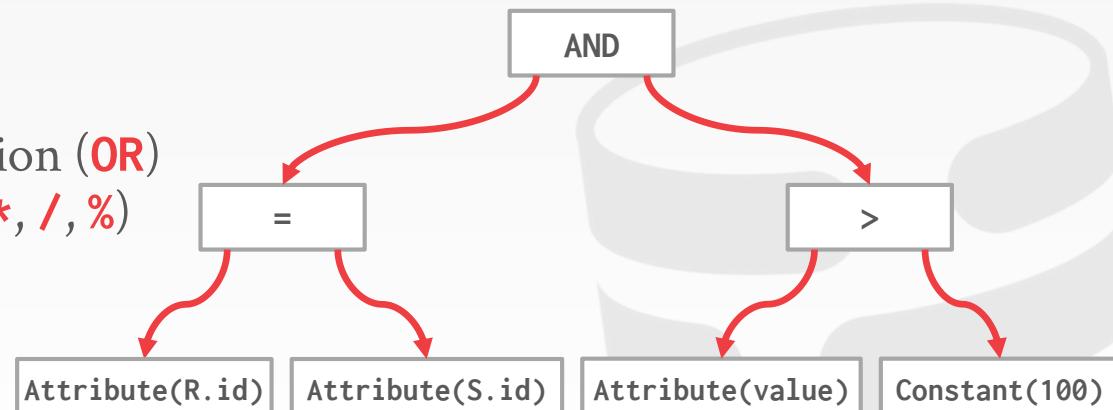
EXPRESSION EVALUATION

The DBMS represents a **WHERE** clause as an **expression tree**.

The nodes in the tree represent different expression types:

- Comparisons ($=, <, >, !=$)
- Conjunction (**AND**), Disjunction (**OR**)
- Arithmetic Operators ($+, -, *, /, \%$)
- Constant Values
- Tuple Attribute References

```
SELECT R.id, S.cdate
  FROM R JOIN S
 WHERE S.value > 100
```



EXPRESSION EVALUATION

```
SELECT * FROM S  
WHERE B.value = ? + 1
```



EXPRESSION EVALUATION

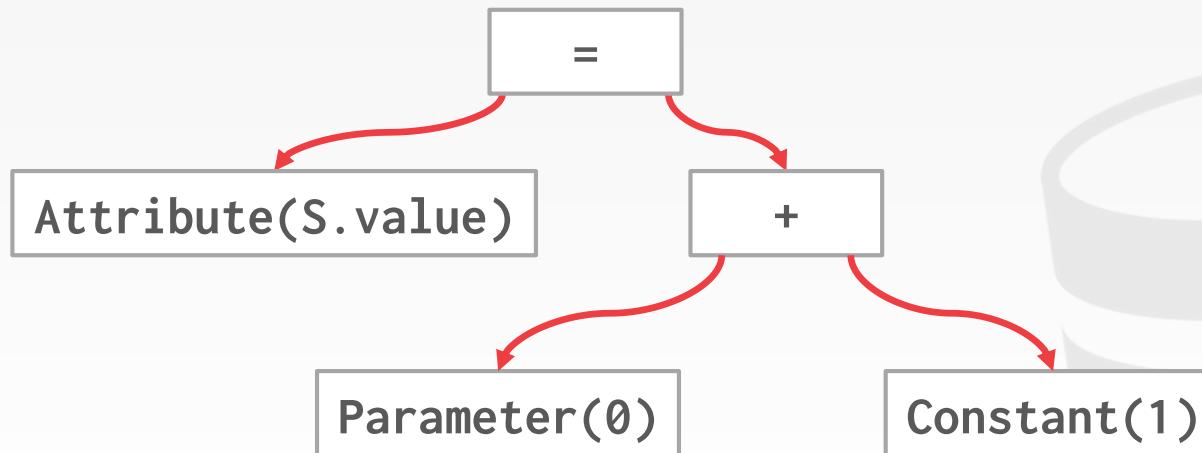
Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple
(123, 1000)

Query Parameters
(int:999)

Table Schema
S→(int:id, int:value)



EXPRESSION EVALUATION

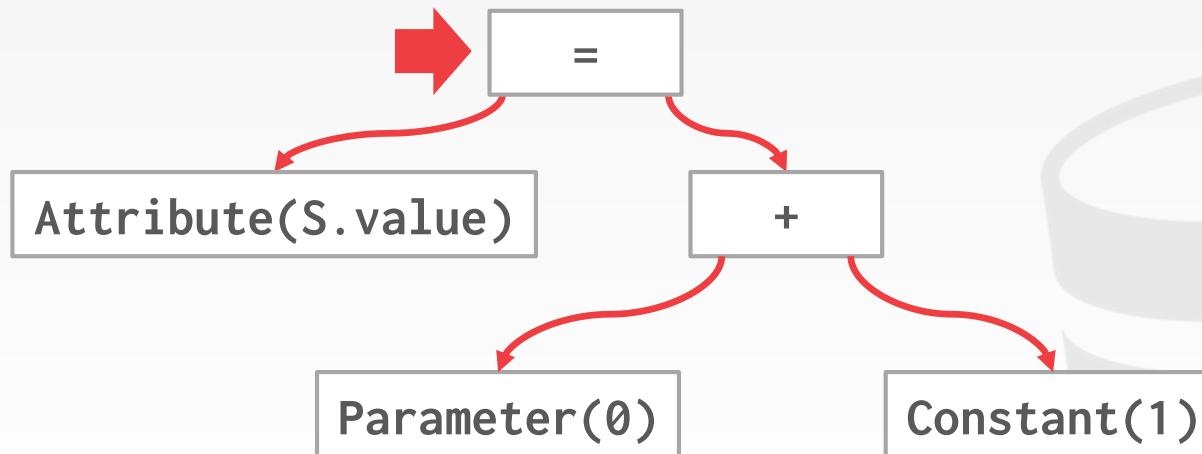
Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple
(123, 1000)

Query Parameters
(int:999)

Table Schema
S→(int:id, int:value)

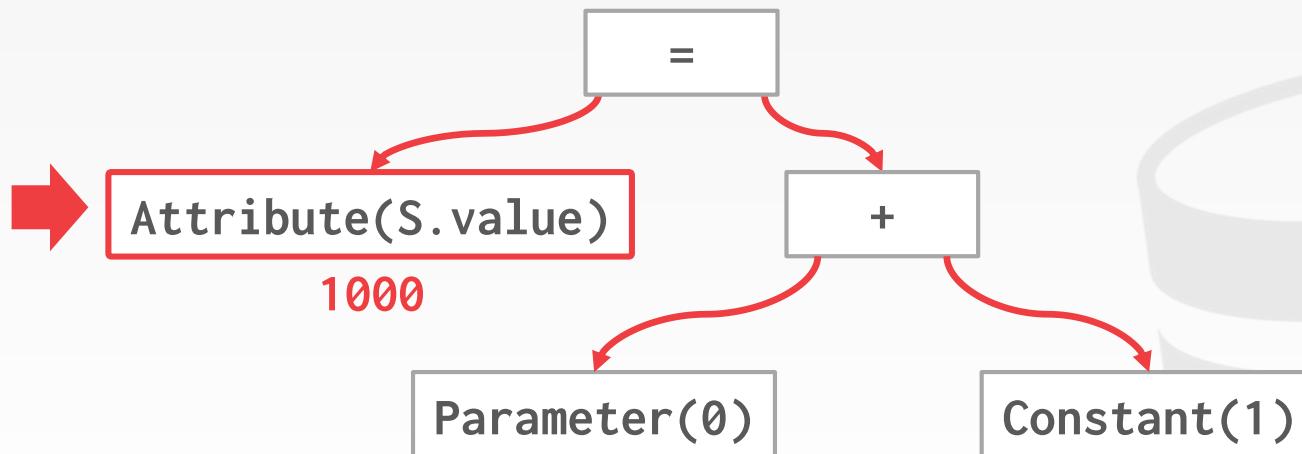


EXPRESSION EVALUATION

Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple (123, 1000)	Query Parameters (int:999)	Table Schema S→(int:id, int:value)
------------------------------	-------------------------------	---------------------------------------



EXPRESSION EVALUATION

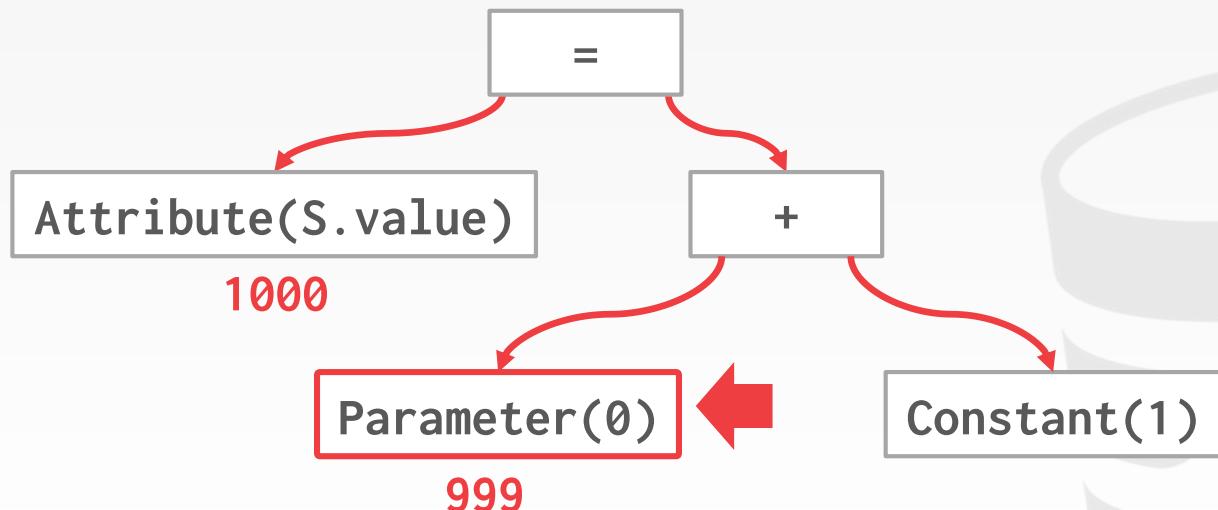
Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple
(123, 1000)

Query Parameters
(int:999)

Table Schema
S→(int:id, int:value)

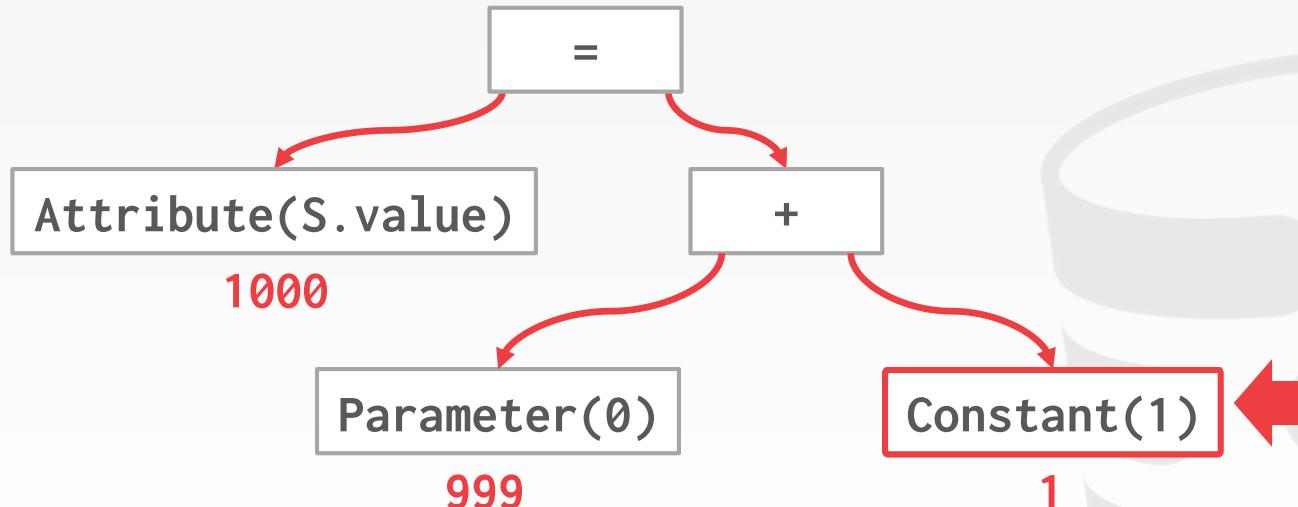


EXPRESSION EVALUATION

Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple (123, 1000)	Query Parameters (int:999)	Table Schema S→(int:id, int:value)
------------------------------	-------------------------------	---------------------------------------

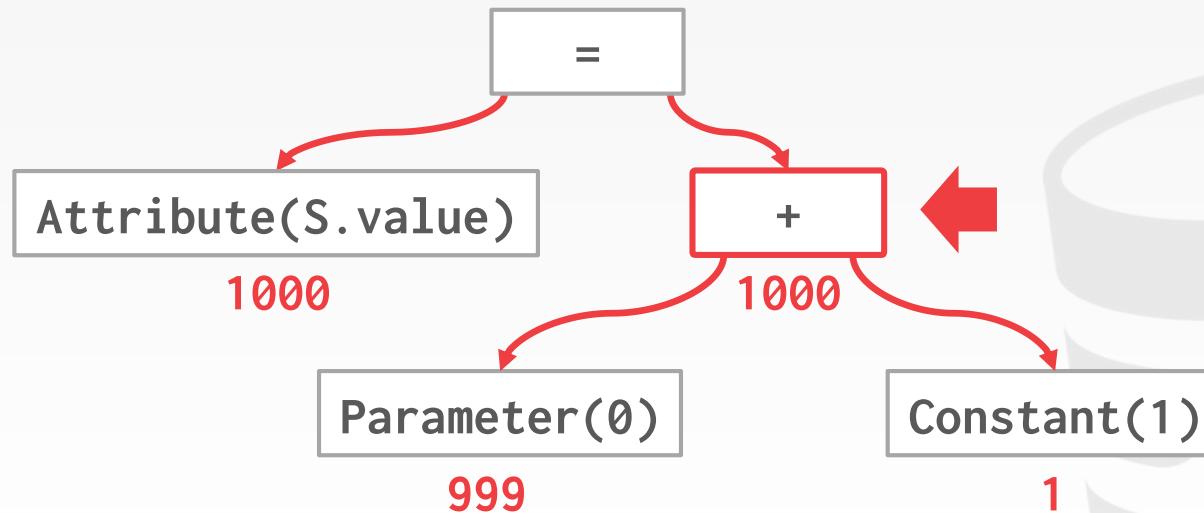


EXPRESSION EVALUATION

Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple (123, 1000)	Query Parameters (int:999)	Table Schema S→(int:id, int:value)
------------------------------	-------------------------------	---------------------------------------

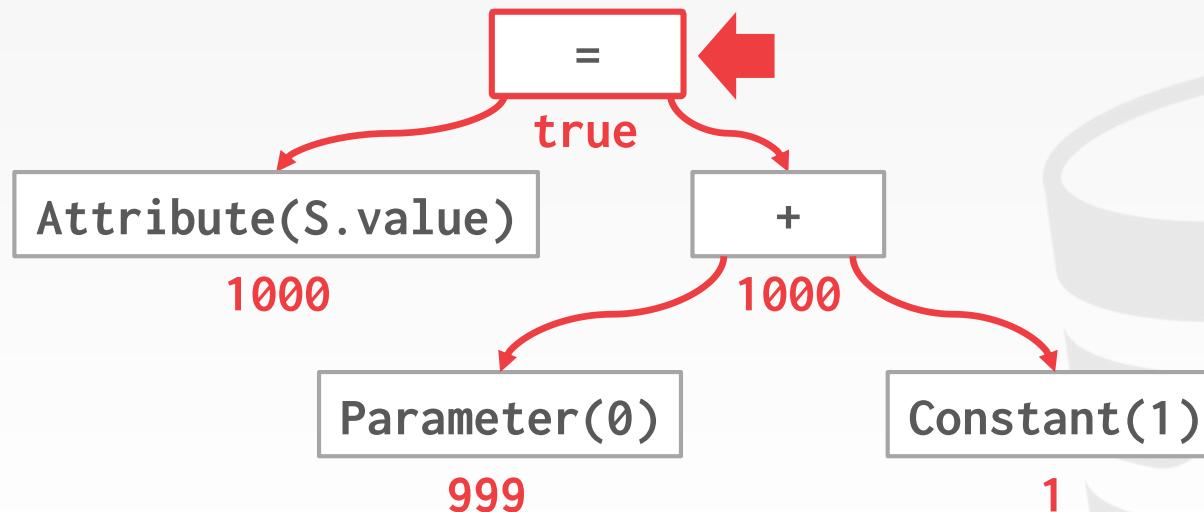


EXPRESSION EVALUATION

Execution Context

```
SELECT * FROM S
WHERE B.value = ? + 1
```

Current Tuple (123, 1000)	Query Parameters (int:999)	Table Schema S→(int:id, int:value)
------------------------------	-------------------------------	---------------------------------------



EXPRESSION EVALUATION

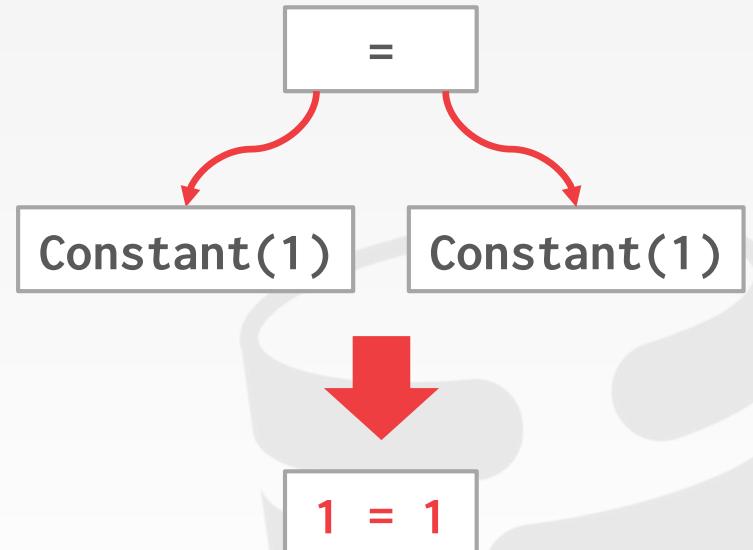
Evaluating predicates in this manner is slow.

- The DBMS traverses the tree and for each node that it visits it must figure out what the operator needs to do.

Consider the predicate "**WHERE 1=1**"

A better approach is to just evaluate the expression directly.

- Think JIT compilation



CONCLUSION

The same query plan can be executed in multiple ways.

(Most) DBMSs will want to use an index scan as much as possible.

Expression trees are flexible but slow.



NEXT CLASS

Parallel Query Execution



13

Query Execution – Part II



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #3 is due Today @ 11:59pm

Mid-Term Exam is Wed Oct 16th @ 12:00pm

Project #2 is due Sun Oct 20th @ 11:59pm



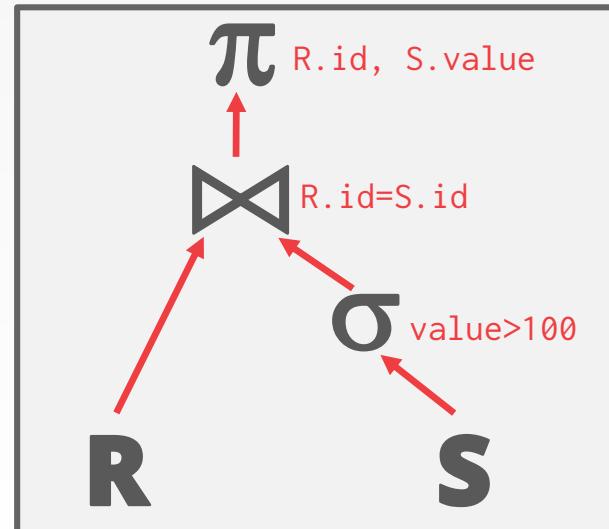
QUERY EXECUTION

We discussed last class how to compose operators together to execute a query plan.

We assumed that the queries execute with a single worker (e.g., thread).

We now need to talk about how to execute with multiple workers...

```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



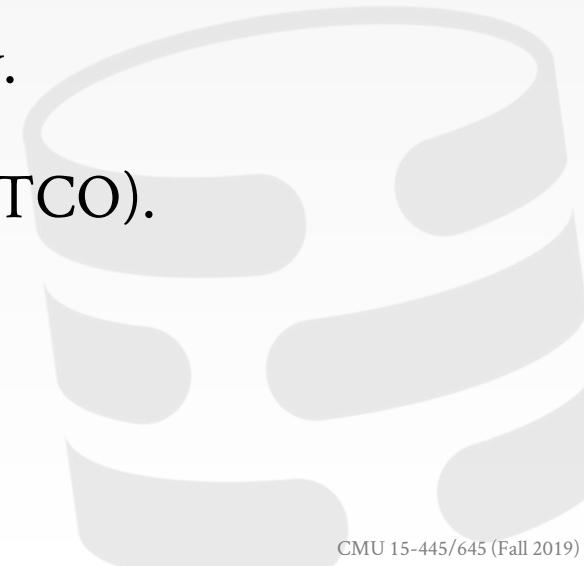
WHY CARE ABOUT PARALLEL EXECUTION?

Increased performance.

- Throughput
- Latency

Increased responsiveness and availability.

Potentially lower *total cost of ownership* (TCO).



PARALLEL VS. DISTRIBUTED

Database is spread out across multiple **resources** to improve different aspects of the DBMS.

Appears as a single database instance to the application.

→ SQL query for a single-resource DBMS should generate same result on a parallel or distributed DBMS.

PARALLEL VS. DISTRIBUTED

Parallel DBMSs:

- Resources are physically close to each other.
- Resources communicate with high-speed interconnect.
- Communication is assumed to cheap and reliable.

Distributed DBMSs:

- Resources can be far from each other.
- Resources communicate using slow(er) interconnect.
- Communication cost and problems cannot be ignored.



TODAY'S AGENDA

Process Models

Execution Parallelism

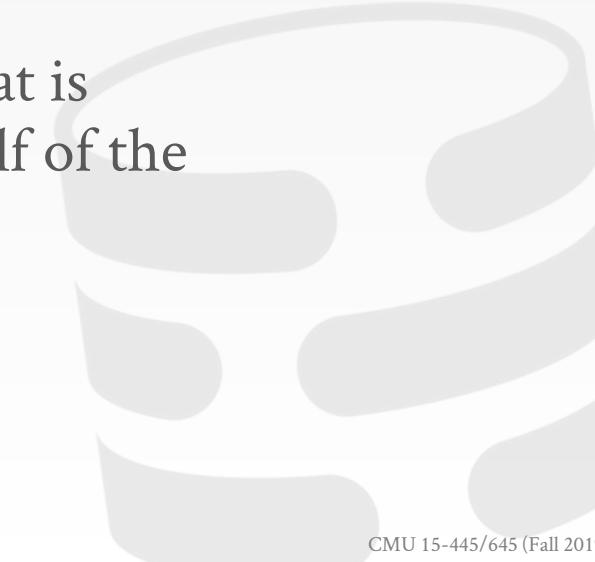
I/O Parallelism



PROCESS MODEL

A DBMS's **process model** defines how the system is architected to support concurrent requests from a multi-user application.

A **worker** is the DBMS component that is responsible for executing tasks on behalf of the client and returning the results.



PROCESS MODELS

Approach #1: Process per DBMS Worker

Approach #2: Process Pool

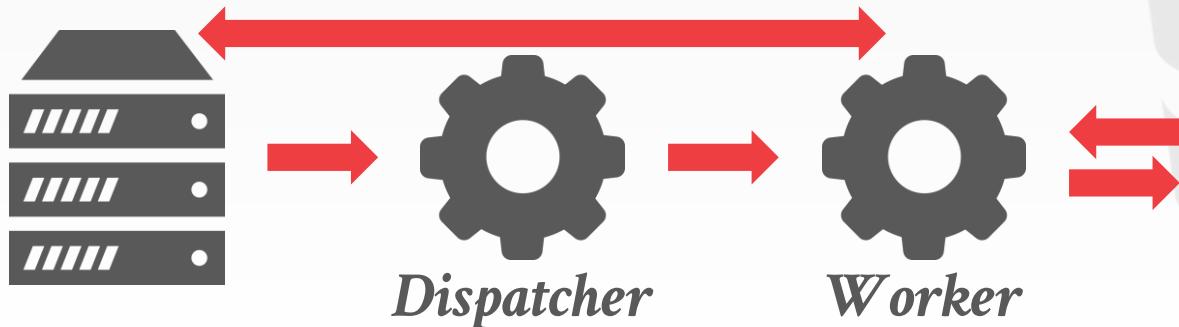
Approach #3: Thread per DBMS Worker



PROCESS PER WORKER

Each worker is a separate OS process.

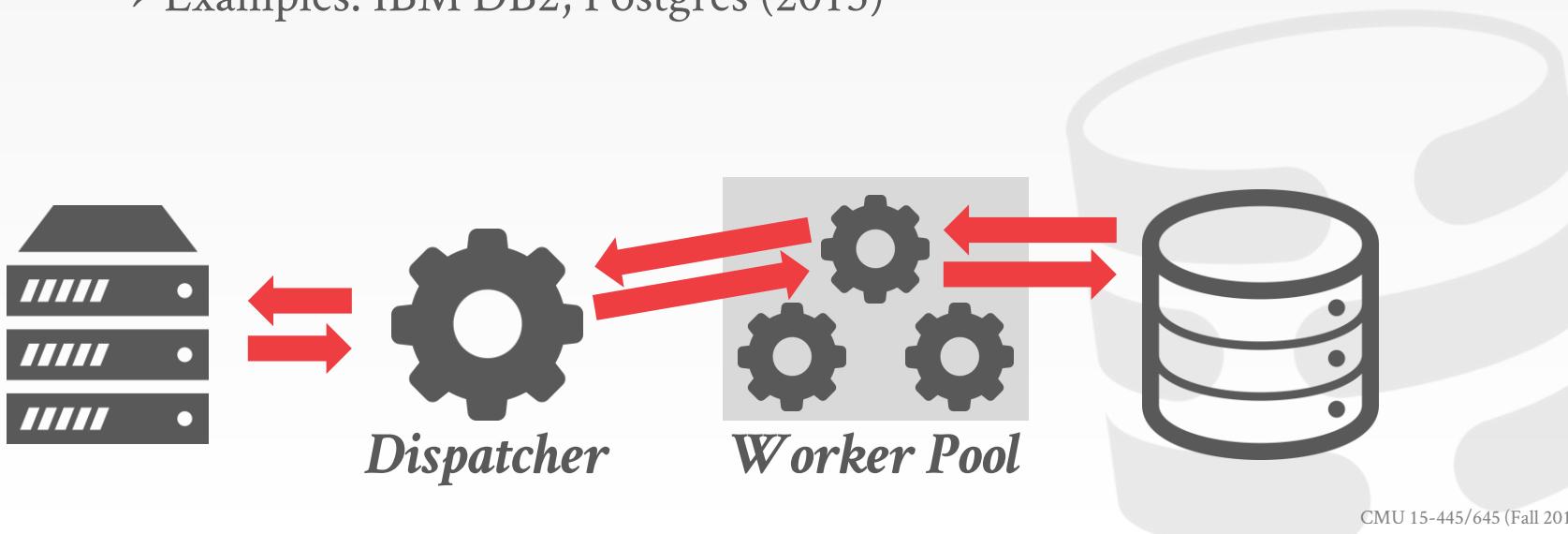
- Relies on OS scheduler.
- Use shared-memory for global data structures.
- A process crash doesn't take down entire system.
- Examples: IBM DB2, Postgres, Oracle



PROCESS POOL



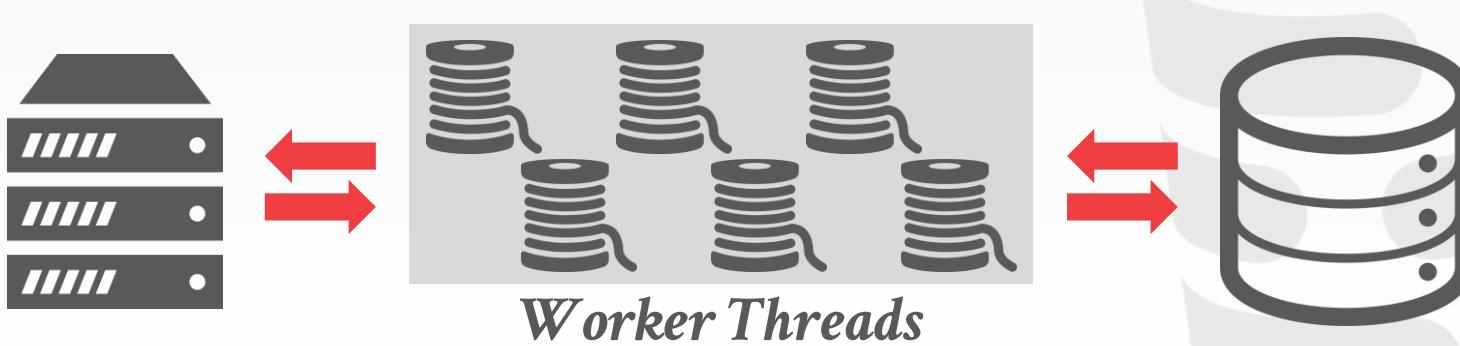
A worker uses any process that is free in a pool
→ Still relies on OS scheduler and shared memory.
→ Bad for CPU cache locality.
→ Examples: IBM DB2, Postgres (2015)



THREAD PER WORKER

Single process with multiple worker threads.

- DBMS manages its own scheduling.
- May or may not use a dispatcher thread.
- Thread crash (may) kill the entire system.
- Examples: IBM DB2, MSSQL, MySQL, Oracle (2014)



PROCESS MODELS

Using a multi-threaded architecture has several advantages:

- Less overhead per context switch.
- Do not have to manage shared memory.

The thread per worker model does not mean that the DBMS supports intra-query parallelism.

Andy is not aware of any new DBMS from last 10 years that doesn't use threads unless they are Postgres forks.

SCHEDULING

For each query plan, the DBMS decides where, when, and how to execute it.

- How many tasks should it use?
- How many CPU cores should it use?
- What CPU core should the tasks execute on?
- Where should a task store its output?

The DBMS *always* knows more than the OS.



INTER- VS. INTRA-QUERY PARALLELISM

Inter-Query: Different queries are executed concurrently.

→ Increases throughput & reduces latency.

Intra-Query: Execute the operations of a single query in parallel.

→ Decreases latency for long-running queries.



INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

If queries are read-only, then this requires little coordination between queries.

Lecture 16

If multiple queries are updating the database at the same time, then this is hard to do correctly...

INTRA-QUERY PARALLELISM

Improve the performance of a single query by executing its operators in parallel.

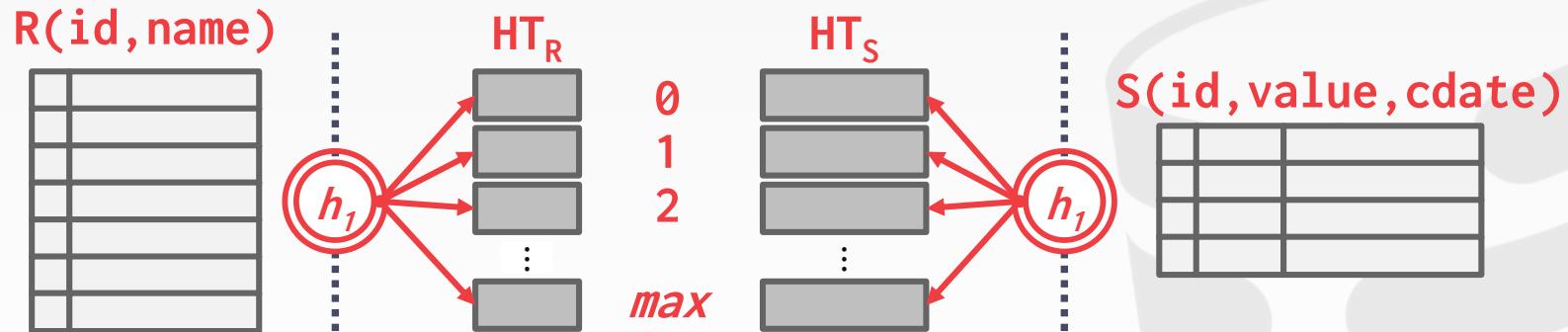
Think of organization of operators in terms of a *producer/consumer* paradigm.

There are parallel algorithms for every relational operator.

→ Can either have multiple threads access centralized data structures or use partitioning to divide work up.

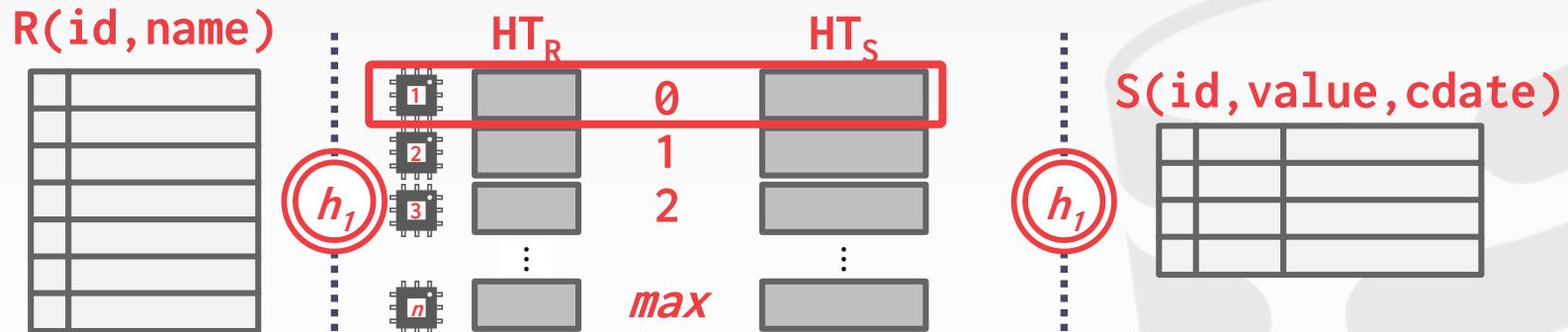
PARALLEL GRACE HASH JOIN

Use a separate worker to perform the join for each level of buckets for **R** and **S** after partitioning.



PARALLEL GRACE HASH JOIN

Use a separate worker to perform the join for each level of buckets for **R** and **S** after partitioning.



INTRA-QUERY PARALLELISM

Approach #1: Intra-Operator (Horizontal)

Approach #2: Inter-Operator (Vertical)

Approach #3: Bushy



INTRA-OPERATOR PARALLELISM

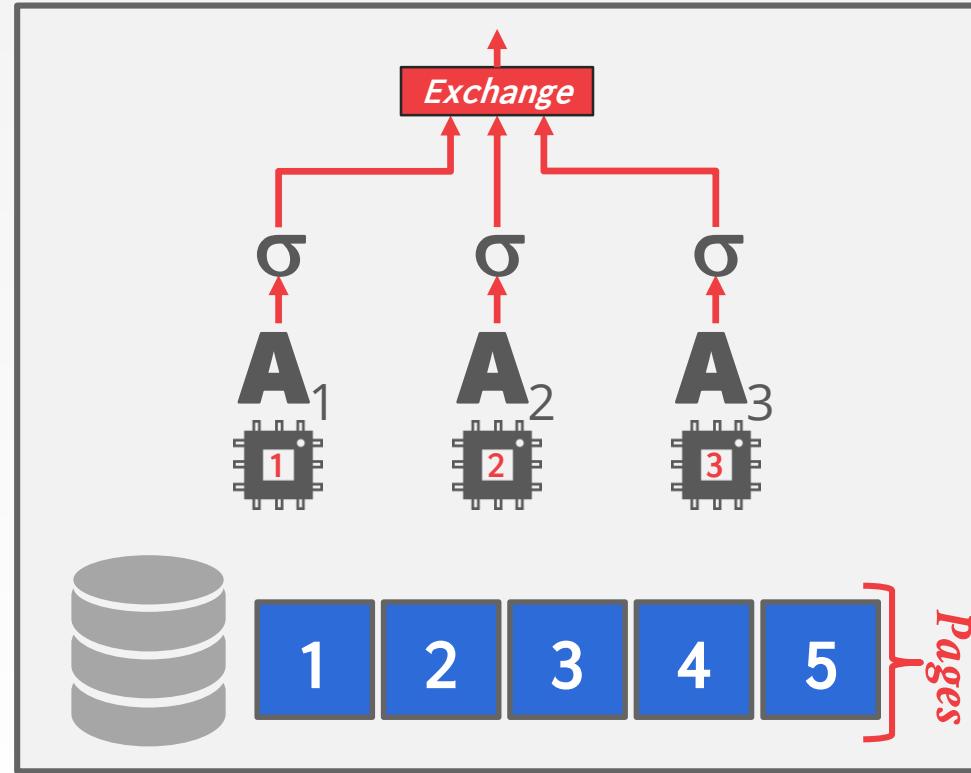
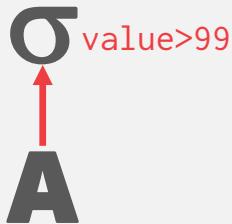
Approach #1: Intra-Operator (Horizontal)

→ Decompose operators into independent fragments that perform the same function on different subsets of data.

The DBMS inserts an exchange operator into the query plan to coalesce results from children operators.

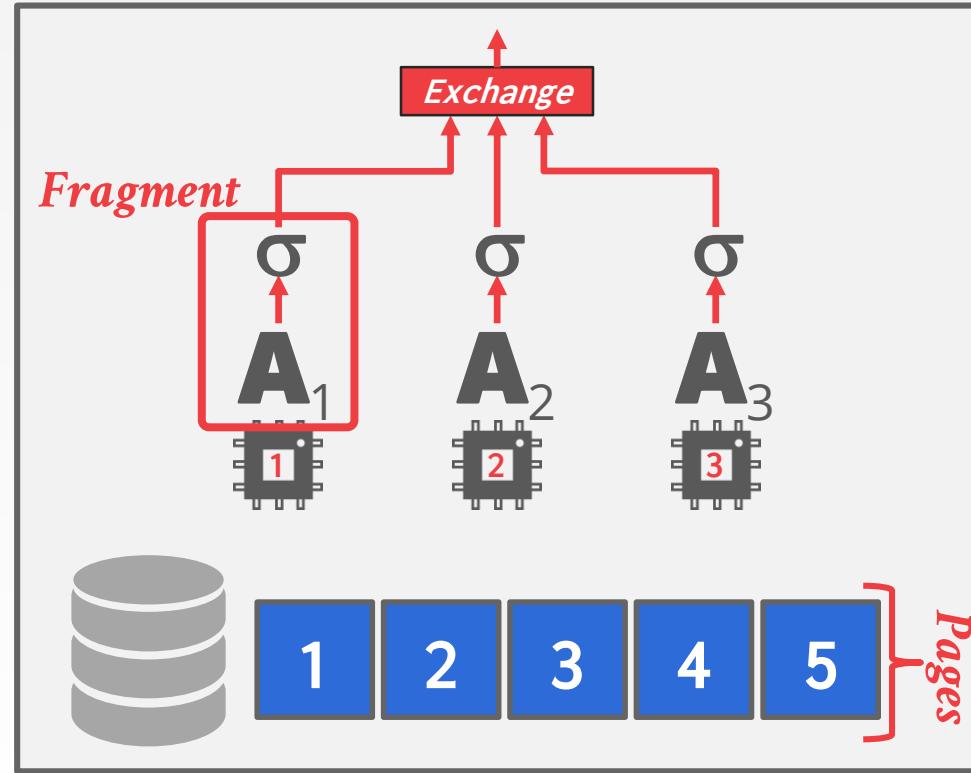
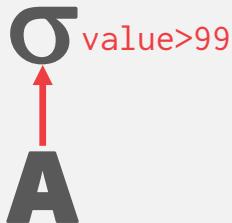
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A  
WHERE A.value > 99
```



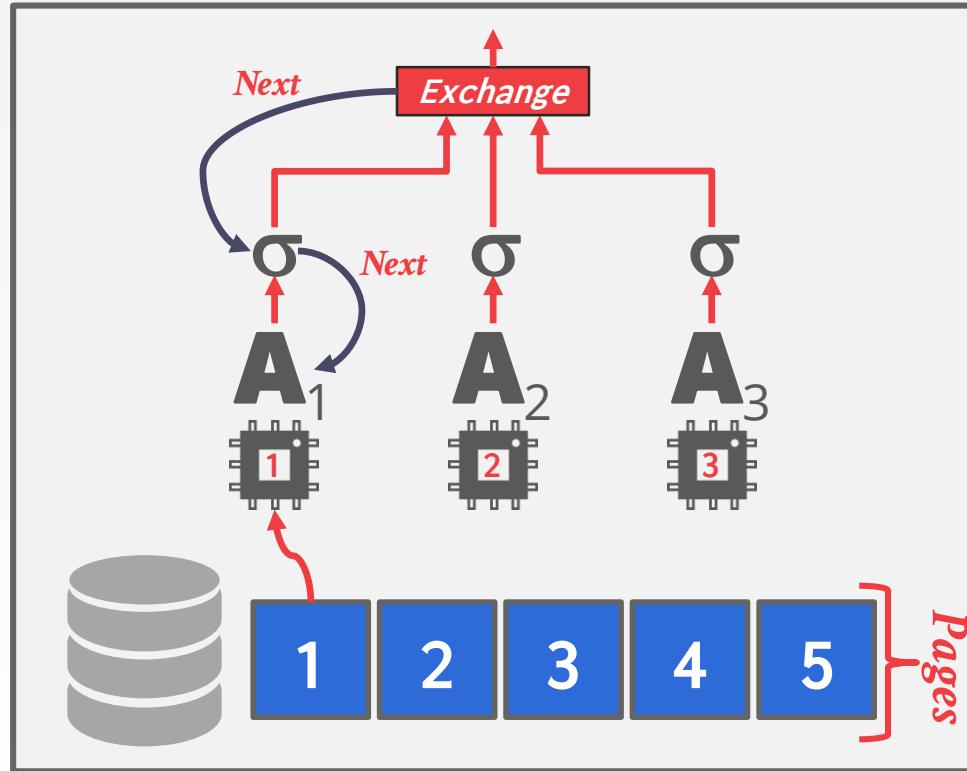
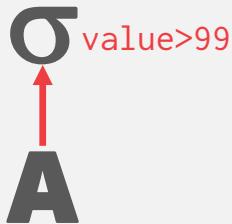
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A  
WHERE A.value > 99
```



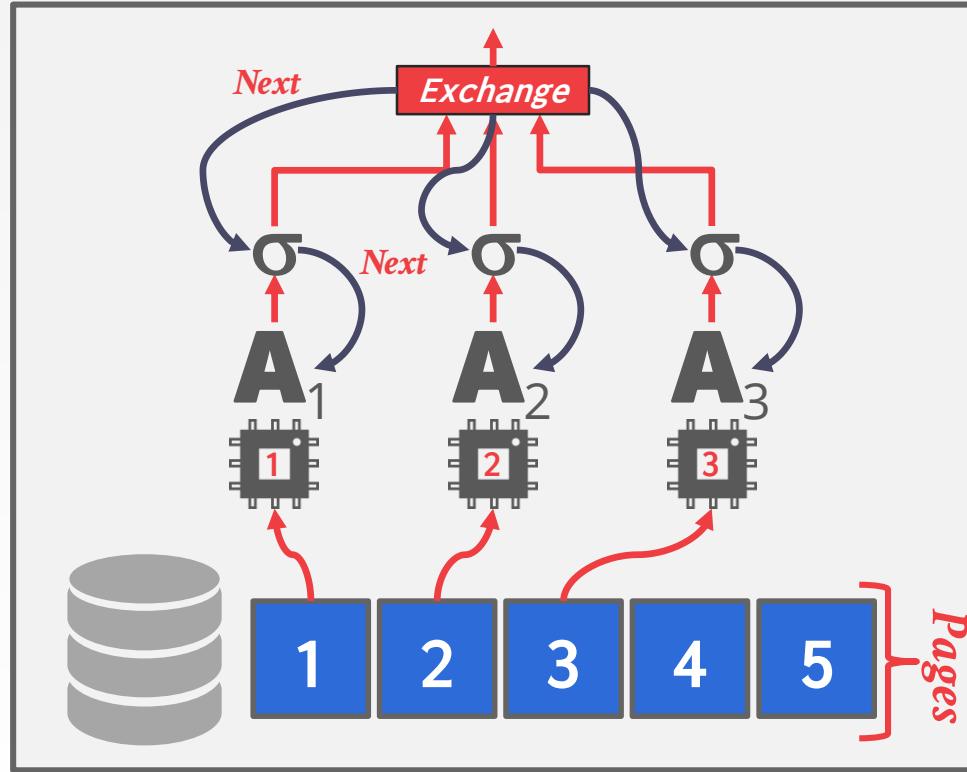
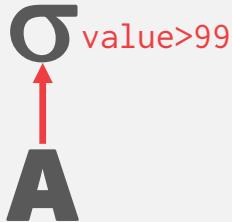
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A  
WHERE A.value > 99
```



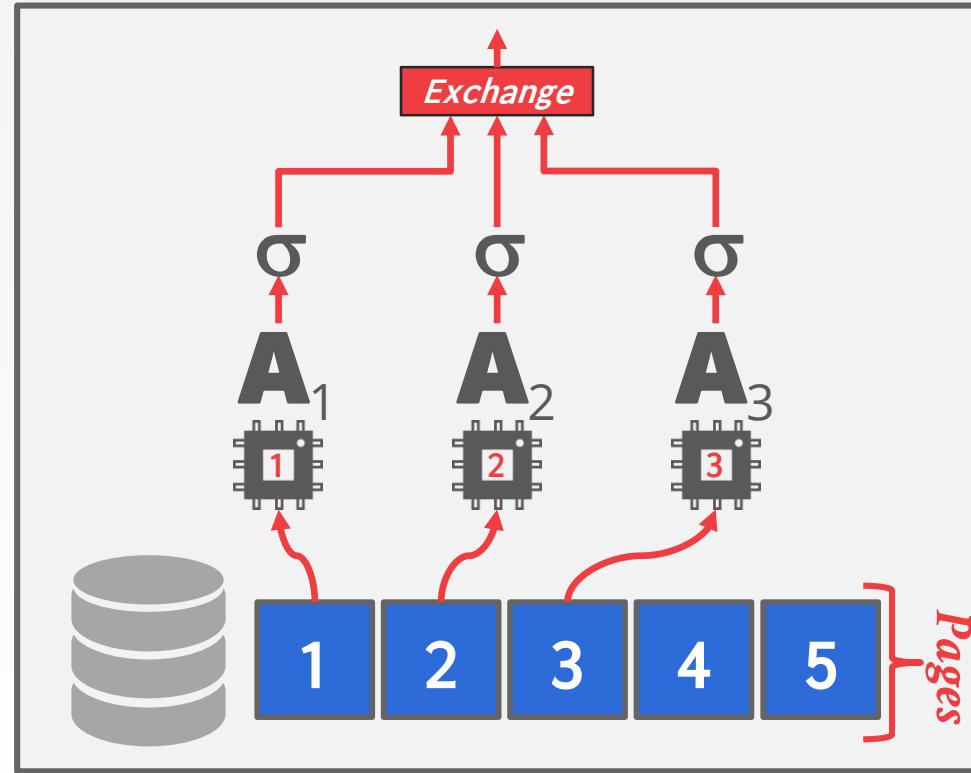
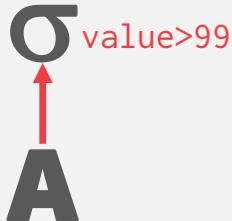
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A  
WHERE A.value > 99
```



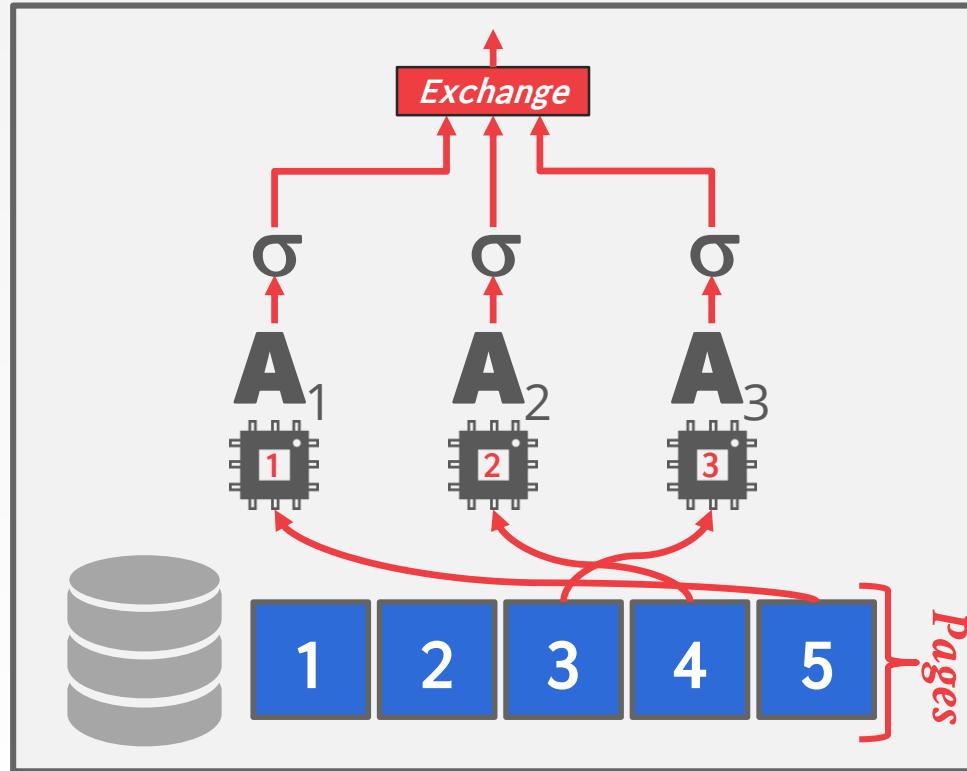
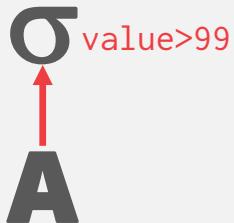
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A  
WHERE A.value > 99
```



INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A  
WHERE A.value > 99
```



EXCHANGE OPERATOR

Exchange Type #1 – Gather

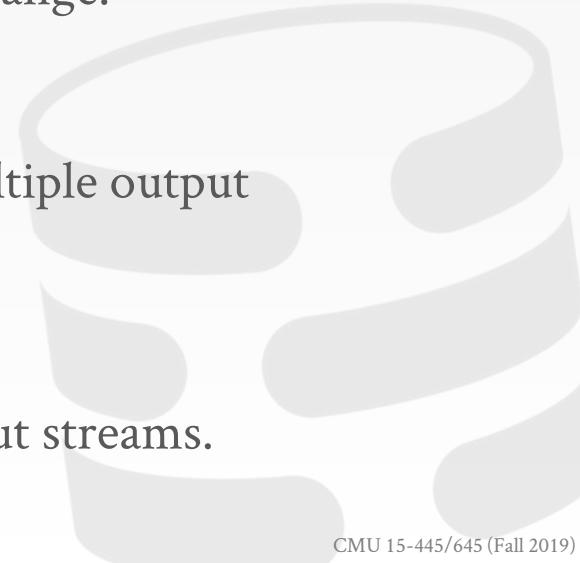
- Combine the results from multiple workers into a single output stream.
- Query plan root must always be a gather exchange.

Exchange Type #2 – Repartition

- Reorganize multiple input streams across multiple output streams.

Exchange Type #3 – Distribute

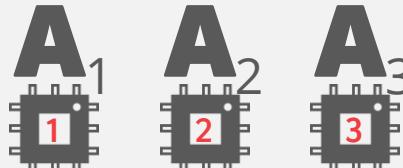
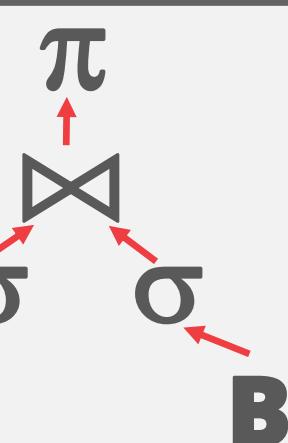
- Split a single input stream into multiple output streams.



Source: [Craig Freedman](#)

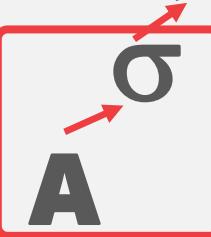
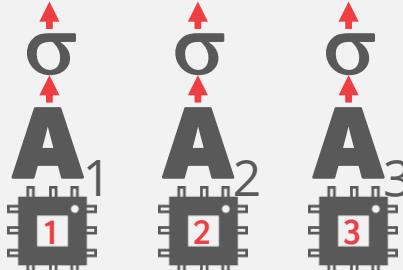
INTRA-OPERATOR PARALLELISM

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

 π  σ σ 

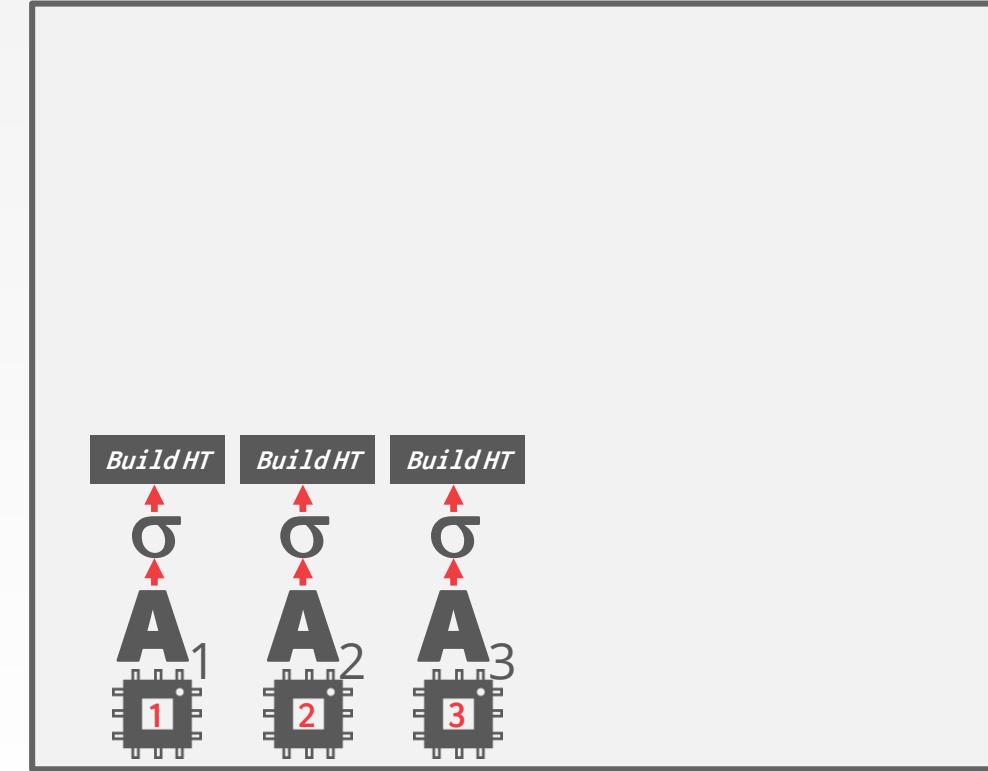
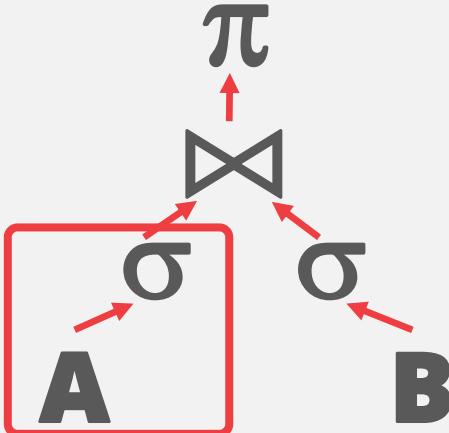
INTRA-OPERATOR PARALLELISM

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

 π  σ  B 

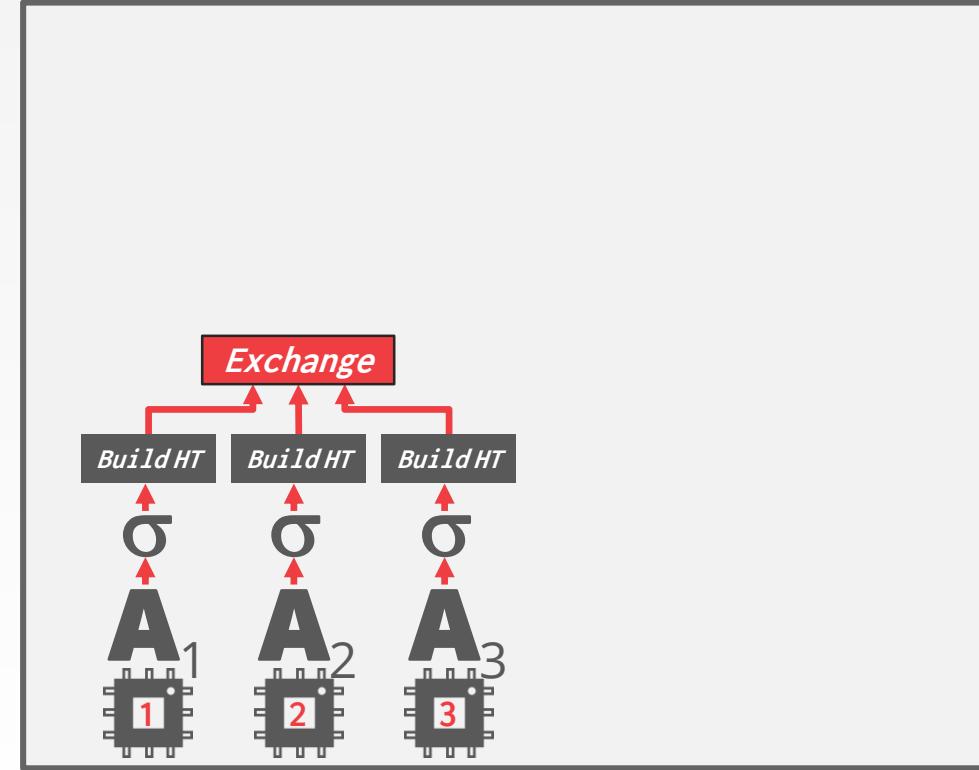
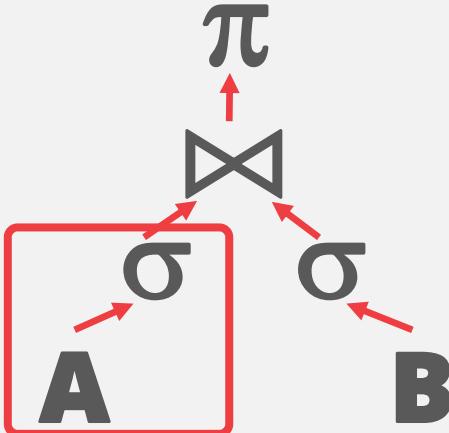
INTRA-OPERATOR PARALLELISM

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



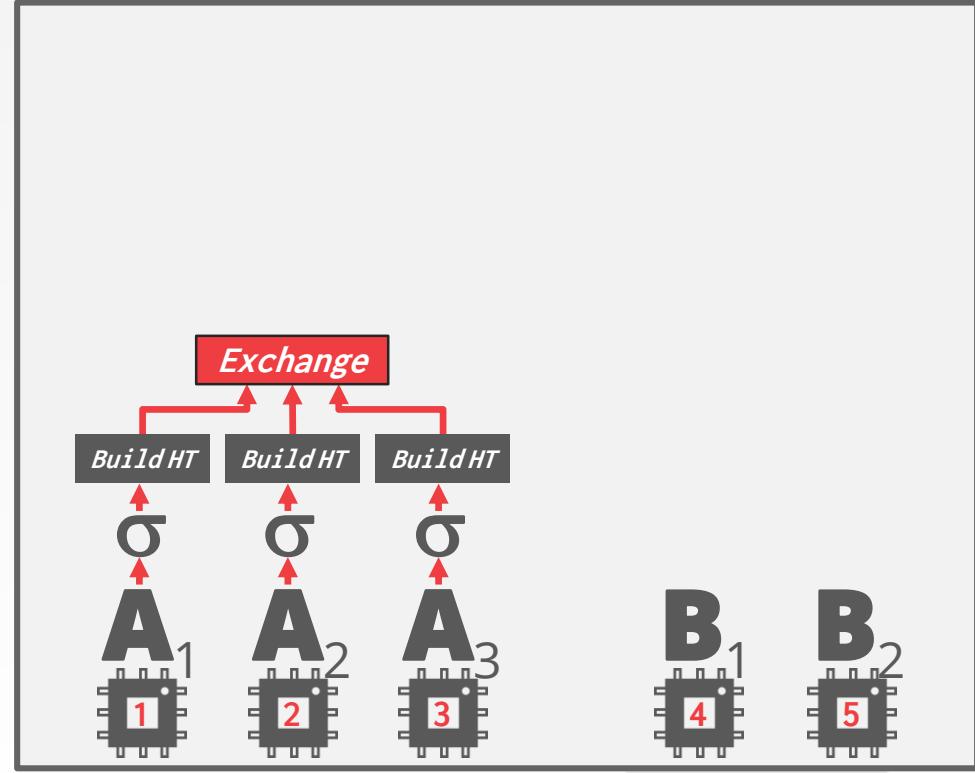
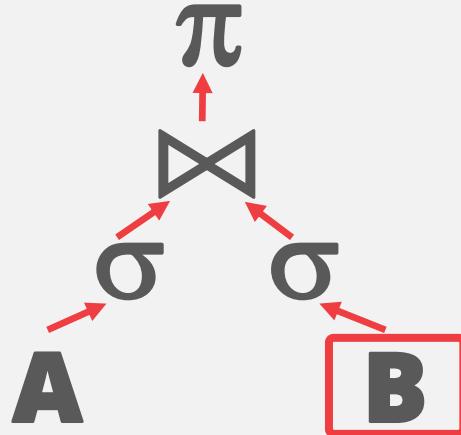
INTRA-OPERATOR PARALLELISM

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



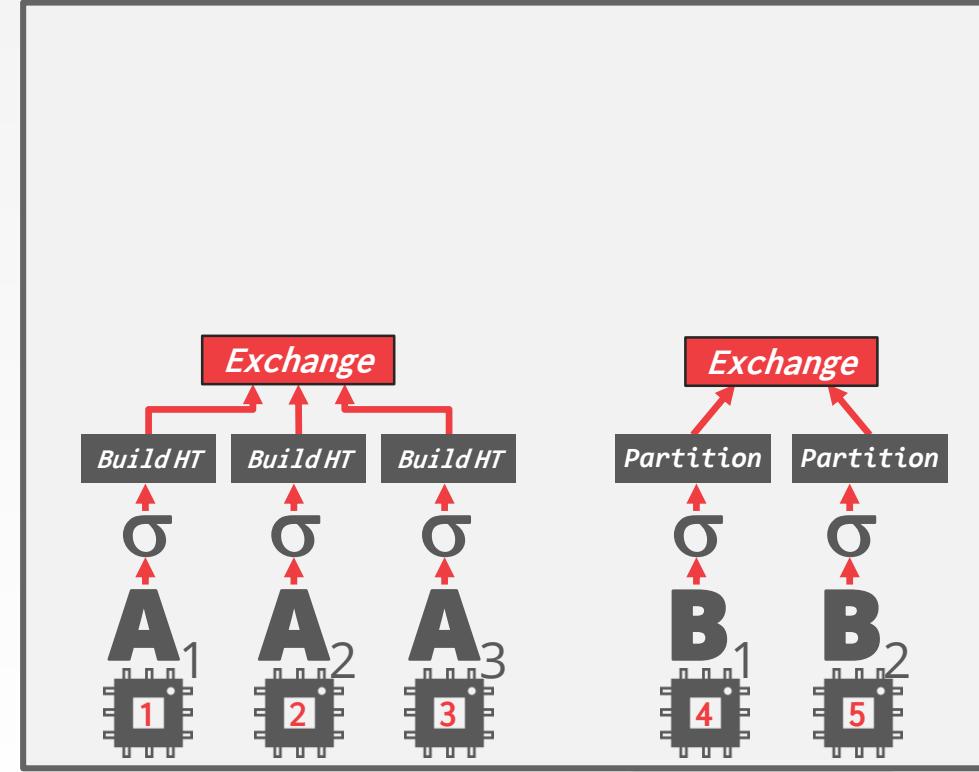
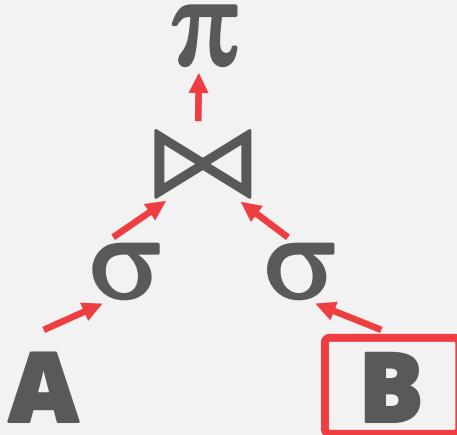
INTRA-OPERATOR PARALLELISM

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



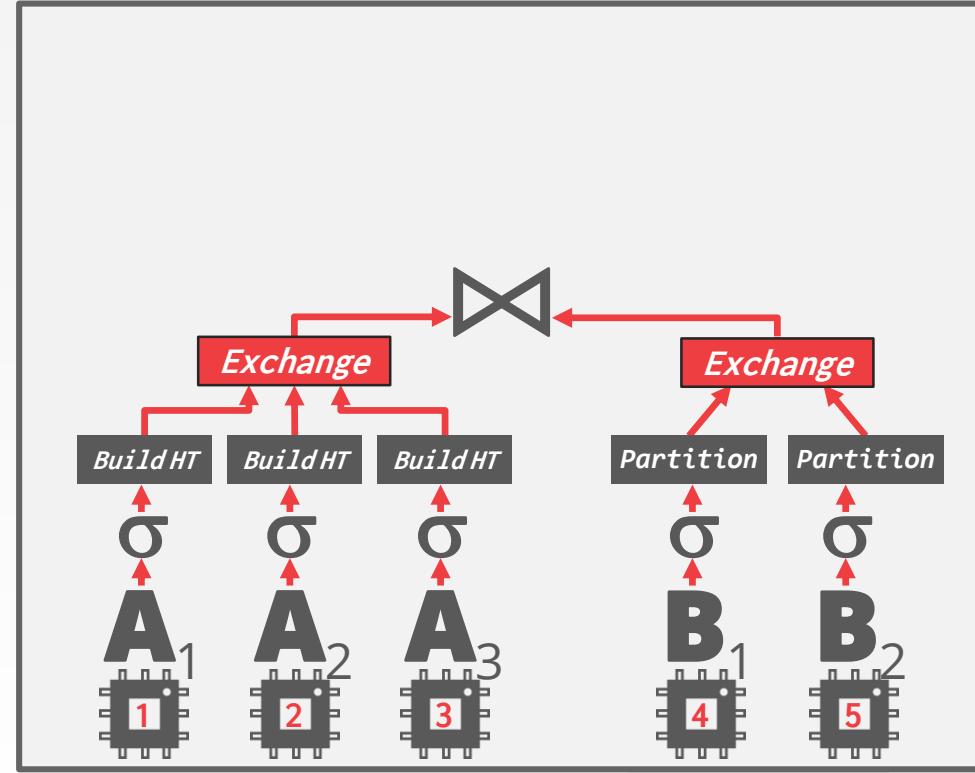
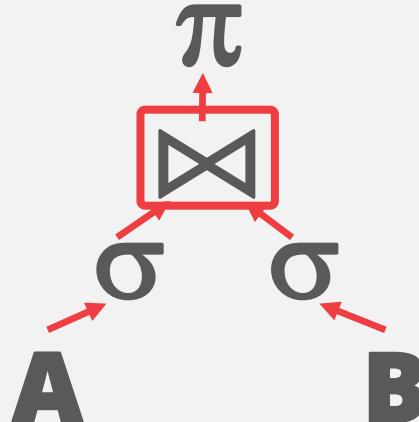
INTRA-OPERATOR PARALLELISM

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



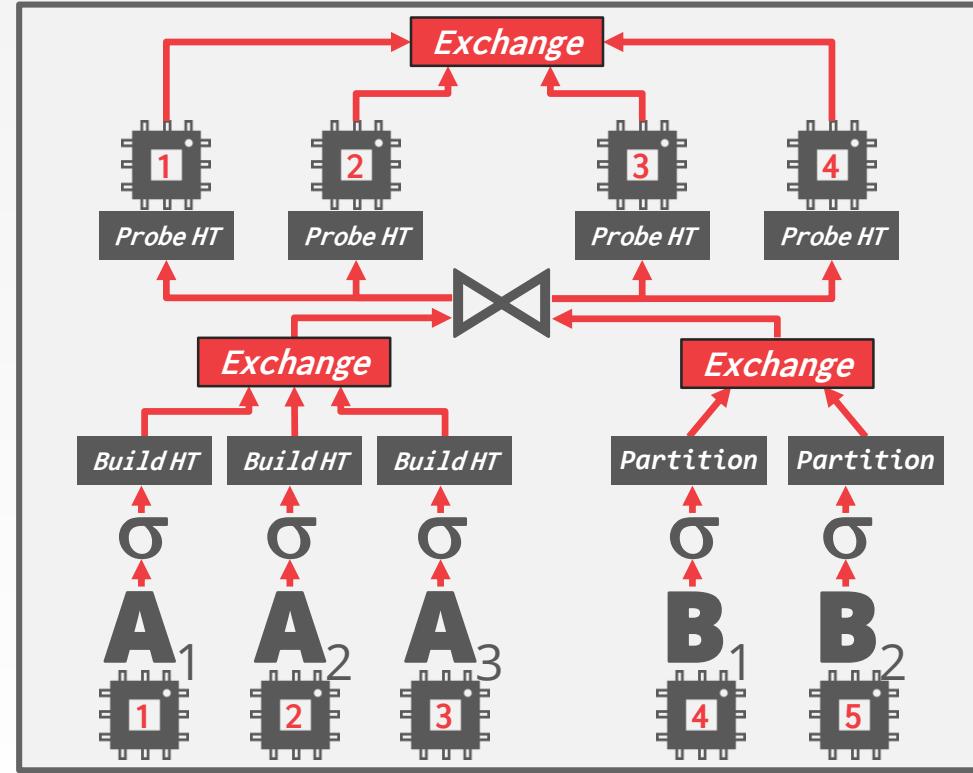
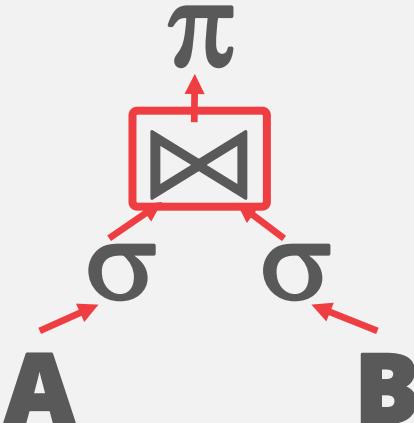
INTRA-OPERATOR PARALLELISM

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



INTRA-OPERATOR PARALLELISM

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



INTER-OPERATOR PARALLELISM

Approach #2: Inter-Operator (Vertical)

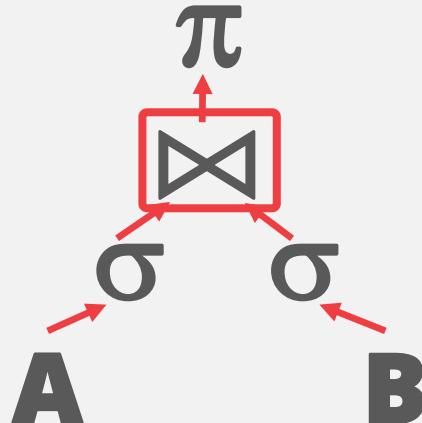
→ Operations are overlapped in order to pipeline data from one stage to the next without materialization.

Also called pipelined parallelism.



INTER-OPERATOR PARALLELISM

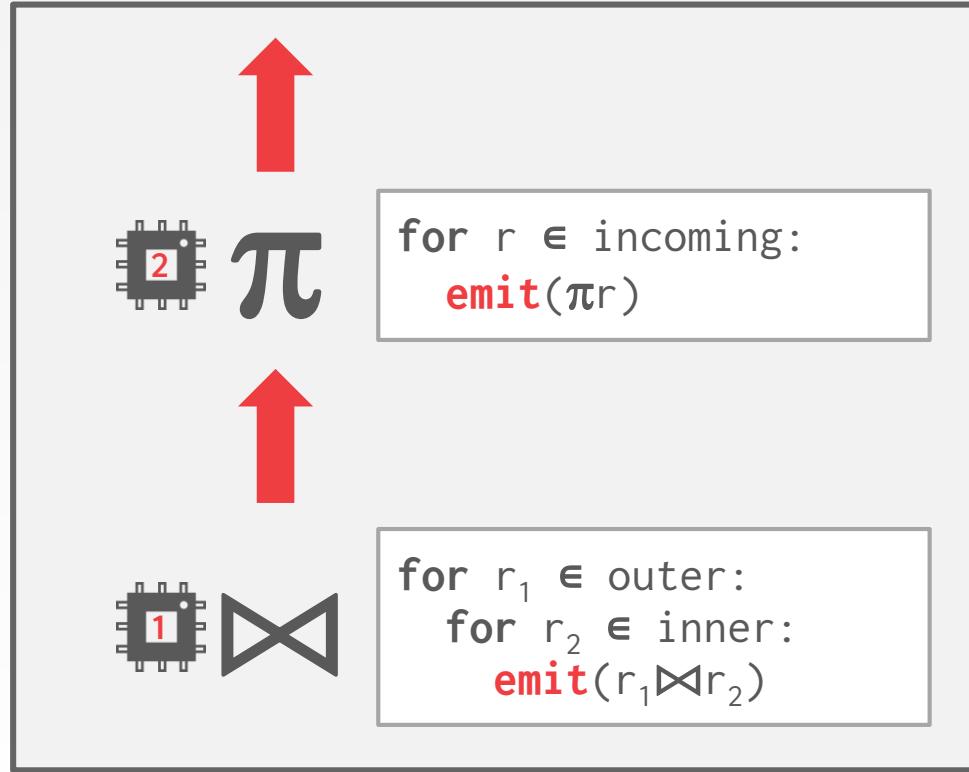
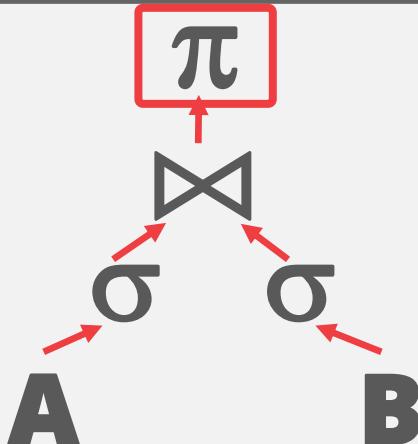
```
SELECT A.id, B.value  
  FROM A JOIN B  
    ON A.id = B.id  
 WHERE A.value < 99  
   AND B.value > 100
```



```
for r1 ∈ outer:  
  for r2 ∈ inner:  
    emit(r1 ⚡ r2)
```

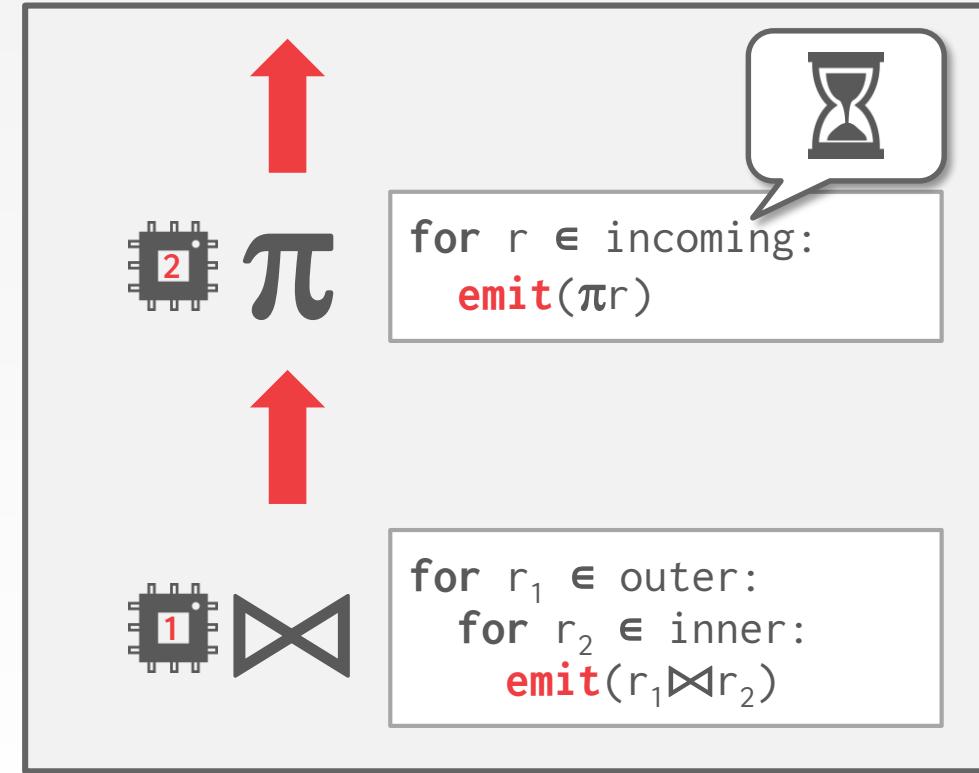
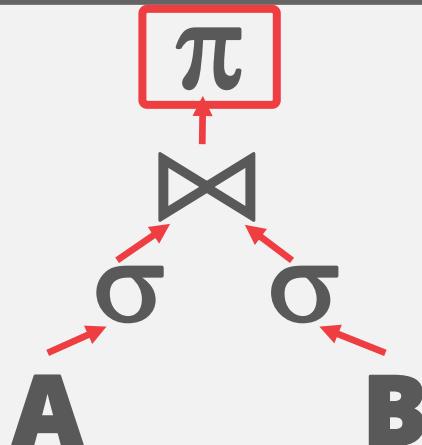
INTER-OPERATOR PARALLELISM

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



INTER-OPERATOR PARALLELISM

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

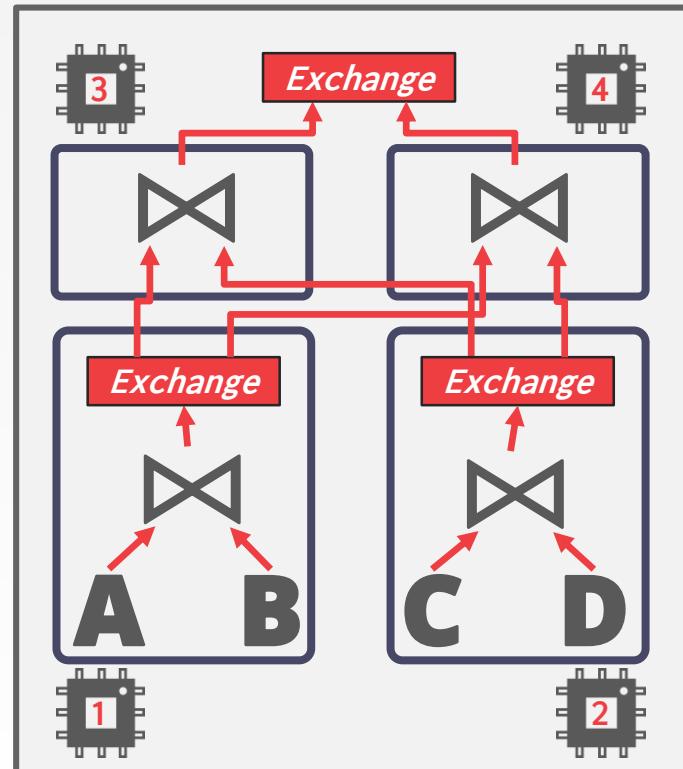


BUSHY PARALLELISM

Approach #3: Bushy Parallelism

- Extension of inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.
- Still need exchange operators to combine intermediate results from segments.

```
SELECT *
  FROM A JOIN B JOIN C JOIN D
```



OBSERVATION

Using additional processes/threads to execute queries in parallel won't help if the disk is always the main bottleneck.

→ Can make things worse if each worker is reading different segments of disk.



I/O PARALLELISM

Split the DBMS installation across multiple storage devices.

- Multiple Disks per Database
- One Database per Disk
- One Relation per Disk
- Split Relation across Multiple Disks

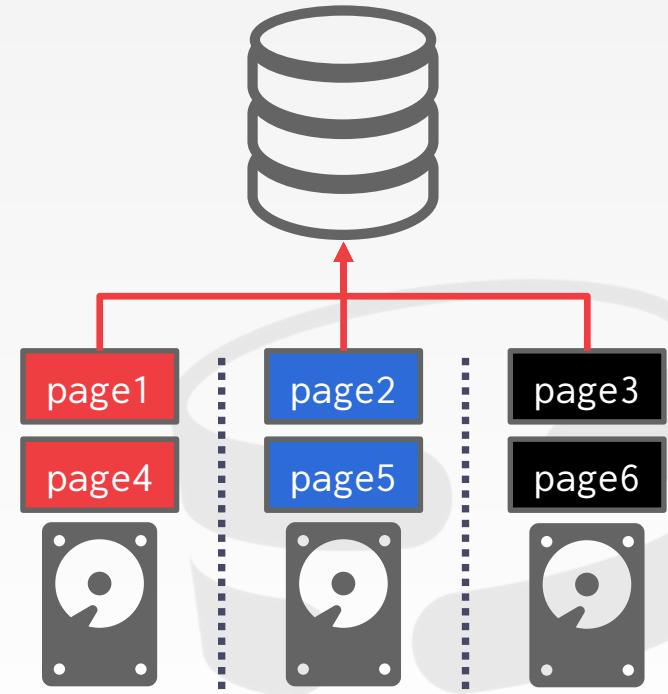


MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.

- Storage Appliances
- RAID Configuration

This is transparent to the DBMS.



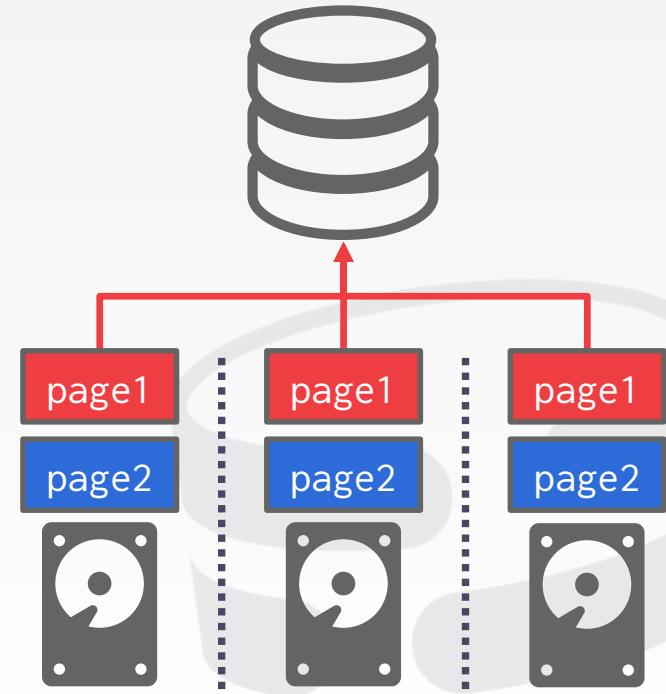
RAID 0 (Stripping)

MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.

- Storage Appliances
- RAID Configuration

This is transparent to the DBMS.



RAID 1 (Mirroring)

DATABASE PARTITIONING

Some DBMSs allow you specify the disk location of each individual database.

→ The buffer pool manager maps a page to a disk location.

This is also easy to do at the filesystem level if the DBMS stores each database in a separate directory.

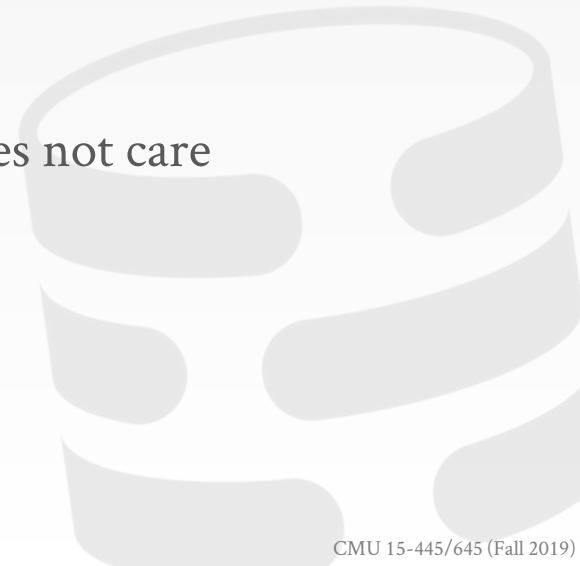
→ The log file might be shared though

PARTITIONING

Split single logical table into disjoint physical segments that are stored/managed separately.

Ideally partitioning is transparent to the application.

- The application accesses logical tables and does not care how things are stored.
- Not always true in distributed DBMSs.



VERTICAL PARTITIONING

Store a table's attributes in a separate location (e.g., file, disk volume).

Have to store tuple information to reconstruct the original record.

```
CREATE TABLE foo (
    attr1 INT,
    attr2 INT,
    attr3 INT,
    attr4 TEXT
);
```

Tuple#1	attr1	attr2	attr3	attr4
Tuple#2	attr1	attr2	attr3	attr4
Tuple#3	attr1	attr2	attr3	attr4
Tuple#4	attr1	attr2	attr3	attr4



VERTICAL PARTITIONING

Store a table's attributes in a separate location (e.g., file, disk volume).

Have to store tuple information to reconstruct the original record.

```
CREATE TABLE foo (
    attr1 INT,
    attr2 INT,
    attr3 INT,
    attr4 TEXT
);
```

Partition #1

Tuple#1	attr1	attr2	attr3
Tuple#2	attr1	attr2	attr3
Tuple#3	attr1	attr2	attr3
Tuple#4	attr1	attr2	attr3

Partition #2

Tuple#1	attr4
Tuple#2	attr4
Tuple#3	attr4
Tuple#4	attr4

HORIZONTAL PARTITIONING

Divide the tuples of a table up into disjoint segments based on some partitioning key.

- Hash Partitioning
- Range Partitioning
- Predicate Partitioning

```
CREATE TABLE foo (
    attr1 INT,
    attr2 INT,
    attr3 INT,
    attr4 TEXT
);
```

	attr1	attr2	attr3	attr4
Tuple#1				
Tuple#2				
Tuple#3				
Tuple#4				

HORIZONTAL PARTITIONING

Divide the tuples of a table up into disjoint segments based on some partitioning key.

- Hash Partitioning
- Range Partitioning
- Predicate Partitioning

Partition #1

Tuple#1	attr1	attr2	attr3	attr4
Tuple#2	attr1	attr2	attr3	attr4

```
CREATE TABLE foo (
    attr1 INT,
    attr2 INT,
    attr3 INT,
    attr4 TEXT
);
```

Partition #2

Tuple#3	attr1	attr2	attr3	attr4
Tuple#4	attr1	attr2	attr3	attr4

CONCLUSION

Parallel execution is important.
(Almost) every DBMS support this.

This is really hard to get right.

- Coordination Overhead
- Scheduling
- Concurrency Issues
- Resource Contention



MIDTERM EXAM

Who: You

What: Midterm Exam

When: Wed Oct 16th @ 12:00pm - 1:20pm

Where: MM 103

Why: <https://youtu.be/GHPB1eCROSA>

Covers up to Query Execution II (inclusive).

- Please email Andy if you need special accommodations.
- <https://15445.courses.cs.cmu.edu/fall2019/midterm-guide.html>

MIDTERM EXAM

What to bring:

- CMU ID
- Calculator
- One 8.5x11" page of handwritten notes (double-sided)

What not to bring:

- Live animals
- Your wet laundry
- Votive Candles (aka "Jennifer Lopez" Candles)



RELATIONAL MODEL

Integrity Constraints
Relation Algebra



SQL

Basic operations:

- SELECT / INSERT / UPDATE / DELETE
- WHERE predicates
- Output control

More complex operations:

- Joins
- Aggregates
- Common Table Expressions



STORAGE

Buffer Management Policies

→ LRU / MRU / CLOCK

On-Disk File Organization

→ Heaps

→ Linked Lists

Page Layout

→ Slotted Pages

→ Log-Structured



HASHING

Static Hashing

- Linear Probing
- Robin Hood
- Cuckoo Hashing

Dynamic Hashing

- Extendible Hashing
- Linear Hashing



TREE INDEXES

B+Tree

- Insertions / Deletions
- Splits / Merges
- Difference with B-Tree
- Latch Crabbing / Coupling

Radix Trees



SORTING

Two-way External Merge Sort

General External Merge Sort

Cost to sort different data sets with different number of buffers.



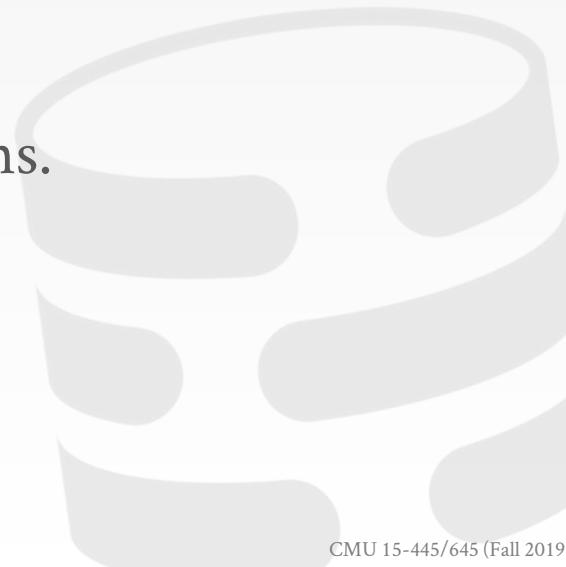
JOINS

Nested Loop Variants

Sort-Merge

Hash

Execution costs under different conditions.



QUERY PROCESSING

Processing Models

→ Advantages / Disadvantages

Parallel Execution

→ Inter- vs. Intra-Operator Parallelism



NEXT CLASS

Query Planning & Optimization



14

Query Planning & Optimization – Part I



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Mid-Term Exam is Wed Oct 16th @ 12:00pm
→ See [mid-term exam guide](#) for more info.

Project #2 is due Sun Oct 20th @ 11:59pm



QUERY OPTIMIZATION

Remember that SQL is declarative.

- User tells the DBMS what answer they want, not how to get the answer.

There can be a big difference in performance based on plan is used:

- See last week: 1.3 hours vs. 0.45 seconds

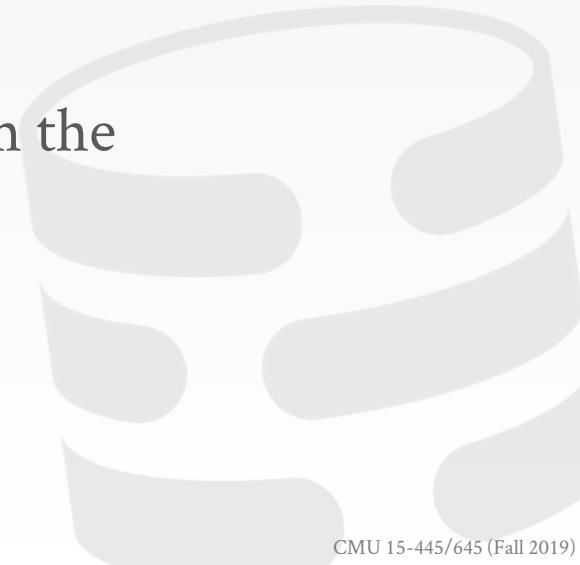


IBM SYSTEM R

First implementation of a query optimizer from the 1970s.

→ People argued that the DBMS could never choose a query plan better than what a human could write.

Many concepts and design decisions from the **System R** optimizer are still used today.



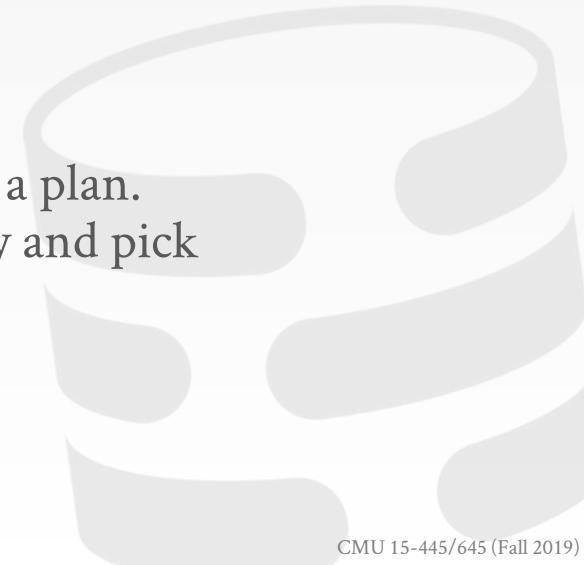
QUERY OPTIMIZATION

Heuristics / Rules

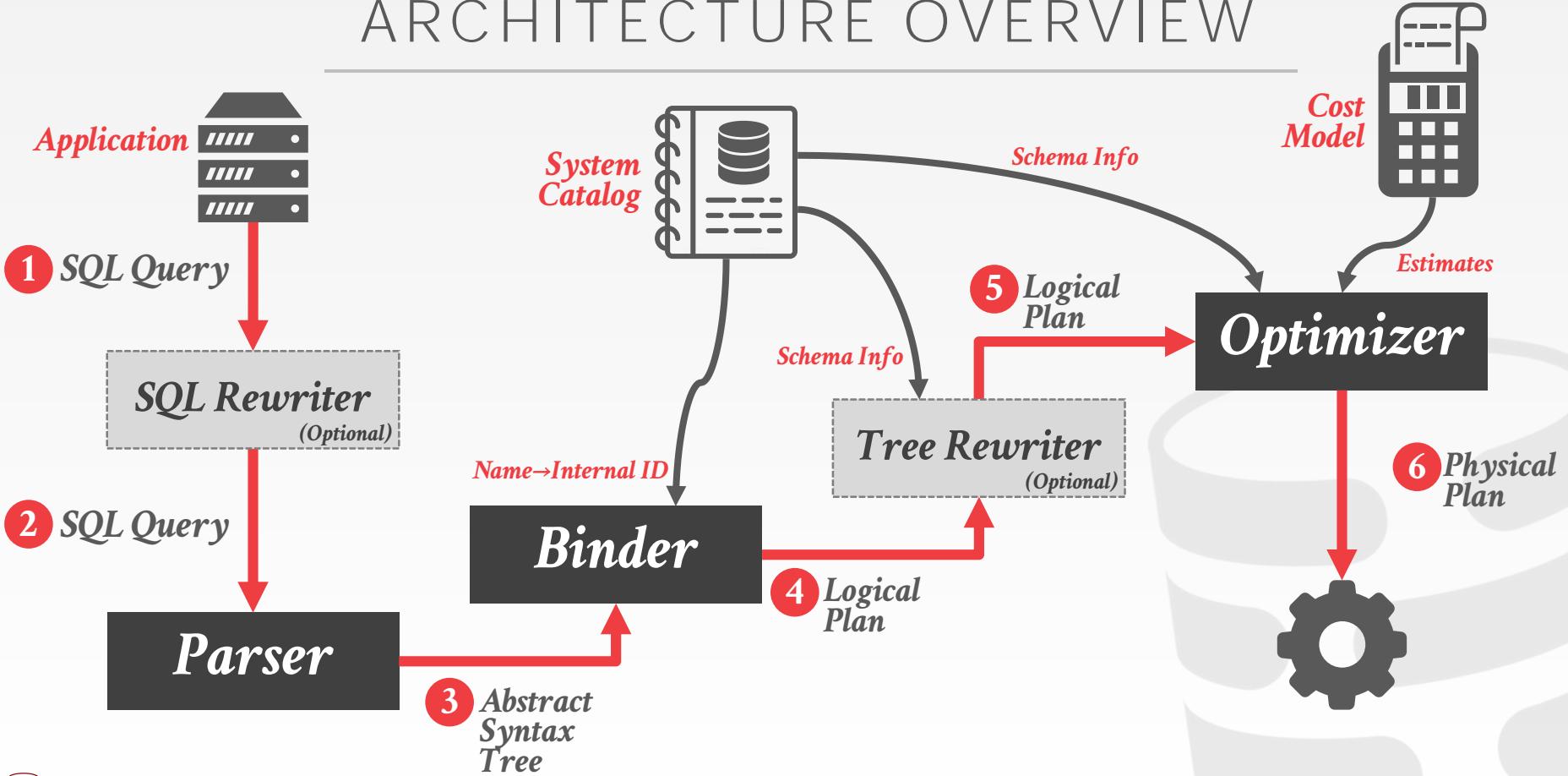
- Rewrite the query to remove stupid / inefficient things.
- These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

- Use a model to estimate the cost of executing a plan.
- Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.



ARCHITECTURE OVERVIEW



LOGICAL VS. PHYSICAL PLANS

The optimizer generates a mapping of a logical algebra expression to the optimal equivalent physical algebra expression.

Physical operators define a specific execution strategy using an access path.

- They can depend on the physical format of the data that they process (i.e., sorting, compression).
- Not always a 1:1 mapping from logical to physical.

QUERY OPTIMIZATION IS NP-HARD

This is the hardest part of building a DBMS.
If you are good at this, you will get paid \$\$\$.

People are starting to look at employing ML to
improve the accuracy and efficacy of optimizers.

I am expanding the Advanced DB Systems class to
cover this topic in greater detail.

TODAY'S AGENDA

Relational Algebra Equivalences
Static Rules



RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions are equivalent if they generate the same set of tuples.

The DBMS can identify better query plans without a cost model.

This is often called query rewriting.



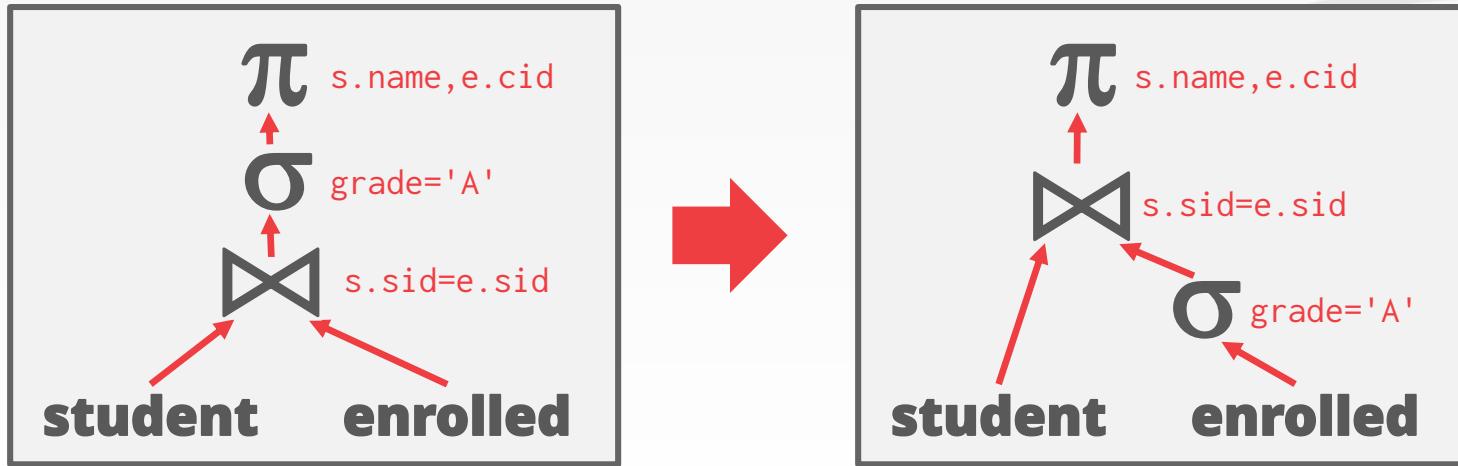
PREDICATE PUSHDOWN

```
SELECT s.name, e.cid  
    FROM student AS s, enrolled AS e  
WHERE s.sid = e.sid  
    AND e.grade = 'A'
```

$$\Pi_{\text{name}, \text{cid}}(\sigma_{\text{grade}='A'}(\text{student} \bowtie \text{enrolled}))$$

PREDICATE PUSHDOWN

```
SELECT s.name, e.cid  
  FROM student AS s, enrolled AS e  
 WHERE s.sid = e.sid  
   AND e.grade = 'A'
```



RELATIONAL ALGEBRA EQUIVALENCES

```
SELECT s.name, e.cid  
FROM student AS s, enrolled AS e  
WHERE s.sid = e.sid  
AND e.grade = 'A'
```

$$\pi_{\text{name}, \text{cid}}(\sigma_{\text{grade}='A'}(\text{student} \bowtie \text{enrolled}))$$

=

$$\pi_{\text{name}, \text{cid}}(\text{student} \bowtie (\sigma_{\text{grade}='A'}(\text{enrolled})))$$

RELATIONAL ALGEBRA EQUIVALENCES

Selections:

- Perform filters as early as possible.
- Reorder predicates so that the DBMS applies the most selective one first.
- Break a complex predicate, and push down

$$\sigma_{p_1 \wedge p_2 \wedge \dots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\dots \sigma_{p_n}(R)))$$

Simplify a complex predicate

$$\rightarrow (X=Y \text{ AND } Y=3) \rightarrow X=3 \text{ AND } Y=3$$



RELATIONAL ALGEBRA EQUIVALENCES

Projections:

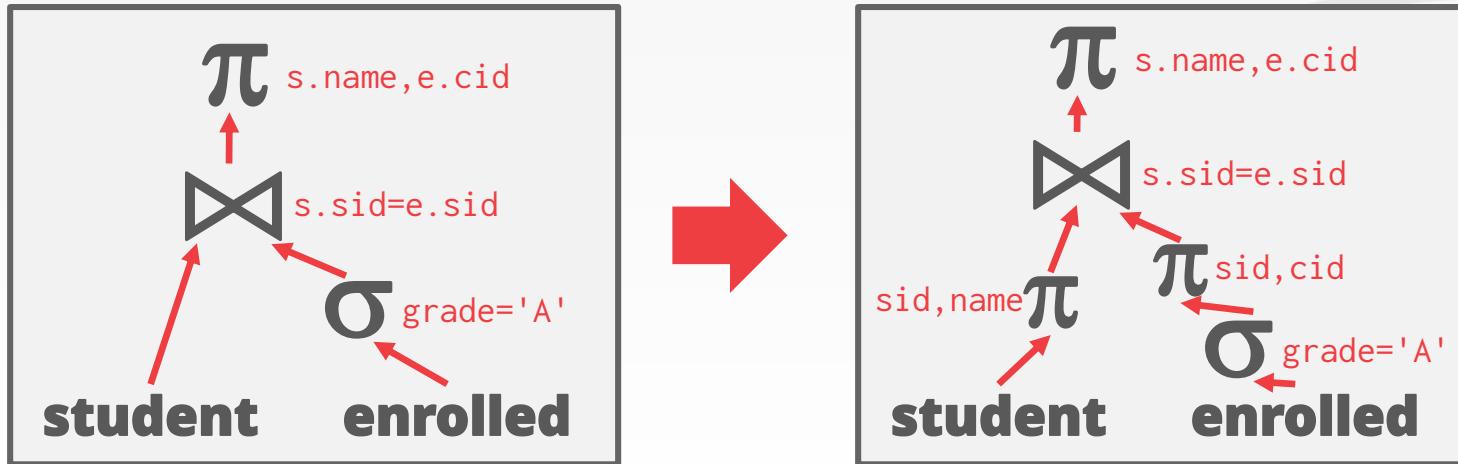
- Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)
- Project out all attributes except the ones requested or required (e.g., joining keys)

This is not important for a column store...



PROJECTION PUSHDOWN

```
SELECT s.name, e.cid
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
    AND e.grade = 'A'
```



```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0; X
```



Source: [Lukas Eder](#)

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0; X
```

```
SELECT * FROM A WHERE 1 = 1;
```

Source: [Lukas Eder](#)

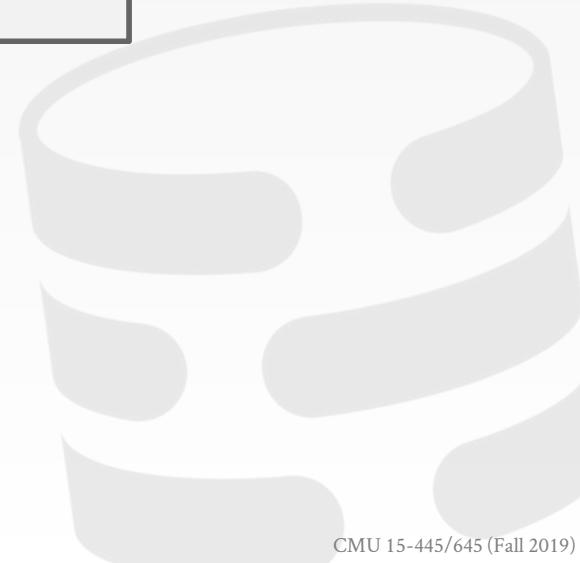
```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0; X
```

```
SELECT * FROM A;
```



Source: [Lukas Eder](#)

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0; X
```

```
SELECT * FROM A;
```

Join Elimination

```
SELECT A1.*  
FROM A AS A1 JOIN A AS A2  
ON A1.id = A2.id;
```

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE 1 = 0; X
```

```
SELECT * FROM A;
```

Join Elimination

```
SELECT * FROM A;
```

Source: [Lukas Eder](#)

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A AS A1  
WHERE EXISTS(SELECT val FROM A AS A2  
            WHERE A1.id = A2.id);
```

Source: [Lukas Eder](#)

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A;
```



Source: [Lukas Eder](#)

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A;
```

Merging Predicates

```
SELECT * FROM A  
WHERE val BETWEEN 1 AND 100  
    OR val BETWEEN 50 AND 150;
```

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A;
```

Merging Predicates

```
SELECT * FROM A  
WHERE val BETWEEN 1 AND 100  
      OR val BETWEEN 50 AND 150;
```

```
CREATE TABLE A (  
    id INT PRIMARY KEY,  
    val INT NOT NULL );
```

MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A;
```

Merging Predicates

```
SELECT * FROM A  
WHERE val BETWEEN 1 AND 150;
```

Source: [Lukas Eder](#)

RELATIONAL ALGEBRA EQUIVALENCES

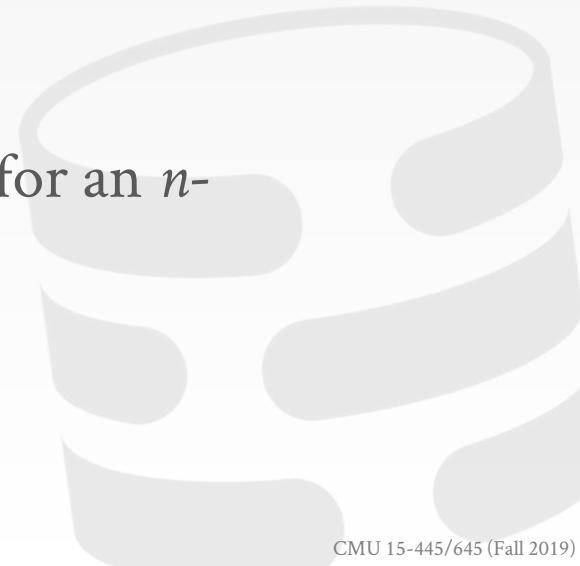
Joins:

→ Commutative, associative

$$R \bowtie S = S \bowtie R$$

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

How many different orderings are there for an n -way join?



RELATIONAL ALGEBRA EQUIVALENCES

How many different orderings are there for an n -way join?

Catalan number $\approx 4^n$

→ Exhaustive enumeration will be too slow.

We'll see in a second how an optimizer limits the search space...



CONCLUSION

We can use static rules and heuristics to optimize a query plan without needing to understand the contents of the database.



NEXT CLASS

MID-TERM EXAM!

→ Seriously, this is not a joke.



15

Query Planning & Optimization – Part 2



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #3 will be released this week.

It is due Sun Nov 17th @ 11:59pm.

Homework #4 will be released next week.

It is due Wed Nov 13th @ 11:59pm.



QUERY OPTIMIZATION

Heuristics / Rules

- Rewrite the query to remove stupid / inefficient things.
- These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

- Use a model to estimate the cost of executing a plan.
- Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

TODAY'S AGENDA

Plan Cost Estimation

Plan Enumeration

Nested Sub-queries



COST ESTIMATION

How long will a query take?

- CPU: Small cost; tough to estimate
- Disk: # of block transfers
- Memory: Amount of DRAM used
- Network: # of messages

How many tuples will be read/written?

It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information...



STATISTICS

The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
Different systems update them at different times.

Manual invocations:

- Postgres/SQLite: **ANALYZE**
- Oracle/MySQL: **ANALYZE TABLE**
- SQL Server: **UPDATE STATISTICS**
- DB2: **RUNSTATS**



STATISTICS

For each relation R , the DBMS maintains the following information:

- N_R : Number of tuples in R .
- $V(A, R)$: Number of distinct values for attribute A .



DERIVABLE STATISTICS

The selection cardinality $SC(A, R)$ is the average number of records with a value for an attribute A given $N_R / V(A, R)$

Note that this assumes *data uniformity*.
→ 10,000 students, 10 colleges – how many students in SCS?

SELECTION STATISTICS

Equality predicates on unique keys are easy to estimate.

```
SELECT * FROM people  
WHERE id = 123
```

```
CREATE TABLE people (  
    id INT PRIMARY KEY,  
    val INT NOT NULL,  
    age INT NOT NULL,  
    status VARCHAR(16)  
);
```

What about more complex predicates? What is their selectivity?

```
SELECT * FROM people  
WHERE val > 1000
```

```
SELECT * FROM people  
WHERE age = 30  
    AND status = 'Lit'
```

COMPLEX PREDICATES

The selectivity (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Formula depends on type of predicate:

- Equality
- Range
- Negation
- Conjunction
- Disjunction



COMPLEX PREDICATES

The selectivity (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Formula depends on type of predicate:

- Equality
- Range
- Negation
- Conjunction
- Disjunction



SELECTIONS – COMPLEX PREDICATES

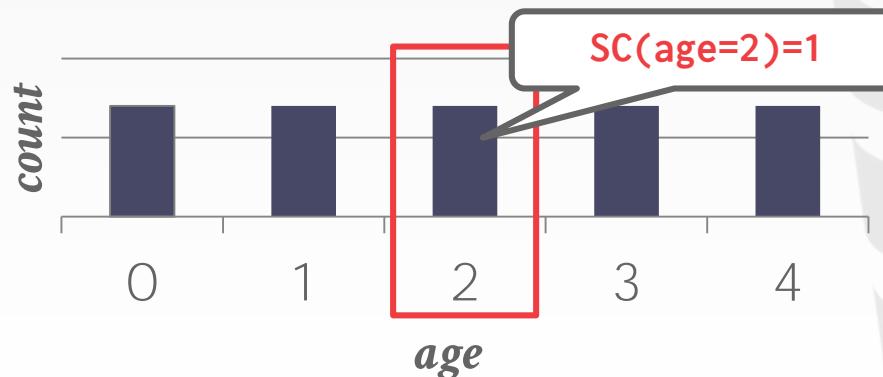
Assume that **V(age, people)** has five distinct values (0–4) and $N_R = 5$

Equality Predicate: A=constant

→ $\text{sel}(A=\text{constant}) = \text{SC}(P) / N_R$

→ Example: $\text{sel}(\text{age}=2) = 1/5$

```
SELECT * FROM people
WHERE age = 2
```



SELECTIONS – COMPLEX PREDICATES

Range Predicate:

- $\text{sel}(A \geq a) = (A_{\max} - a) / (A_{\max} - A_{\min})$
- Example: $\text{sel}(\text{age} \geq 2) \approx (4 - 2) / (4 - 0)$
 $\approx 1/2$

```
SELECT * FROM people
WHERE age >= 2
```

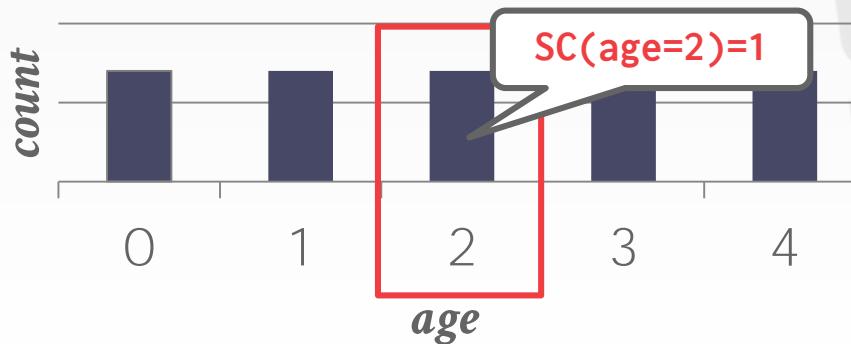


SELECTIONS – COMPLEX PREDICATES

Negation Query:

- $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$
- Example: $\text{sel}(\text{age } \neq 2)$

```
SELECT * FROM people  
WHERE age != 2
```



SELECTIONS – COMPLEX PREDICATES

Negation Query:

→ $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$

→ Example: $\text{sel}(\text{age } \neq 2) = 1 - (1/5) = 4/5$

```
SELECT * FROM people
WHERE age != 2
```

Observation: Selectivity ≈ Probability



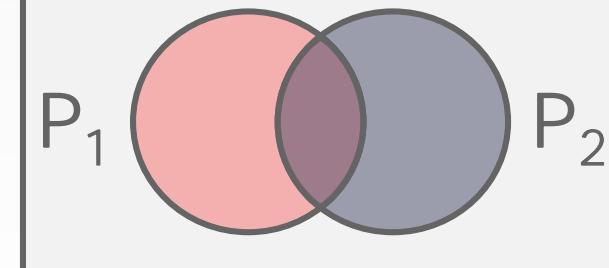
SELECTIONS – COMPLEX PREDICATES

Conjunction:

- $\text{sel}(P1 \wedge P2) = \text{sel}(P1) \bullet \text{sel}(P2)$
- $\text{sel}(\text{age}=2 \wedge \text{name LIKE 'A%'})$

This assumes that the predicates are independent.

```
SELECT * FROM people  
WHERE age = 2  
    AND name LIKE 'A%'
```



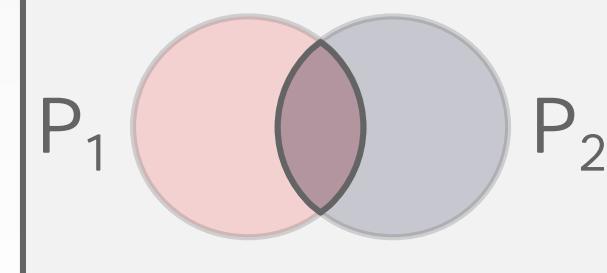
SELECTIONS – COMPLEX PREDICATES

Conjunction:

- $\text{sel}(P1 \wedge P2) = \text{sel}(P1) \bullet \text{sel}(P2)$
- $\text{sel}(\text{age}=2 \wedge \text{name LIKE 'A%'})$

This assumes that the predicates are independent.

```
SELECT * FROM people  
WHERE age = 2  
AND name LIKE 'A%'
```



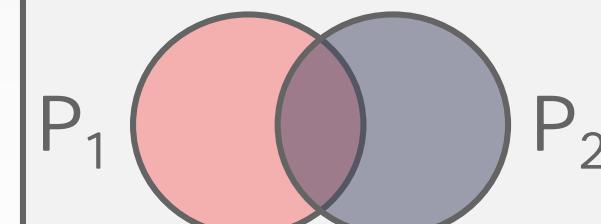
SELECTIONS – COMPLEX PREDICATES

Disjunction:

→ $\text{sel}(P_1 \vee P_2)$
 $= \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \wedge P_2)$
 $= \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \cdot \text{sel}(P_2)$
→ $\text{sel}(\text{age}=2 \text{ OR } \text{name LIKE 'A%'})$

This again assumes that the selectivities are independent.

```
SELECT * FROM people
WHERE age = 2
    OR name LIKE 'A%'
```



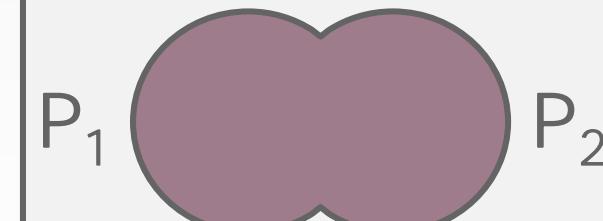
SELECTIONS – COMPLEX PREDICATES

Disjunction:

- $\text{sel}(P_1 \vee P_2)$
- = $\text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \wedge P_2)$
- = $\text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \cdot \text{sel}(P_2)$
- $\text{sel}(\text{age}=2 \text{ OR } \text{name } \text{LIKE } 'A\%')$

This again assumes that the selectivities are independent.

```
SELECT * FROM people
WHERE age = 2
OR name LIKE 'A%'
```



SELECTION CARDINALITY

Assumption #1: Uniform Data

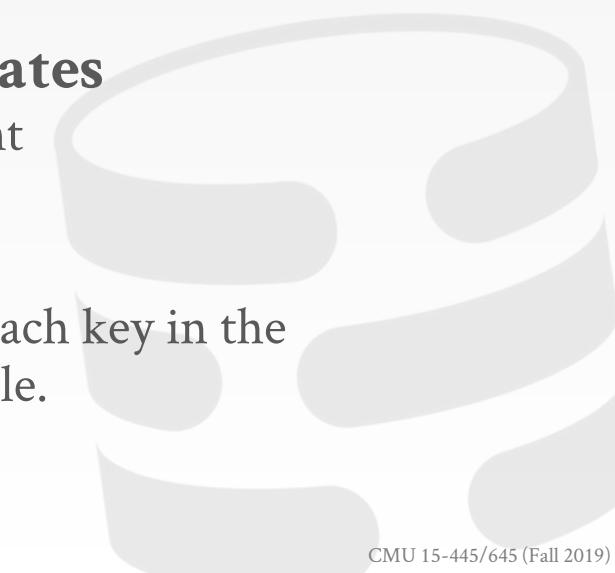
- The distribution of values (except for the heavy hitters) is the same.

Assumption #2: Independent Predicates

- The predicates on attributes are independent

Assumption #3: Inclusion Principle

- The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.



CORRELATED ATTRIBUTES

Consider a database of automobiles:

→ # of Makes = 10, # of Models = 100

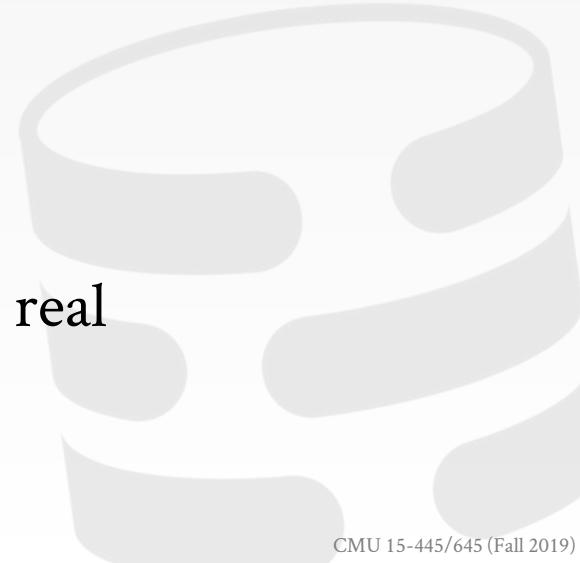
And the following query:

→ `(make="Honda" AND model="Accord")`

With the independence and uniformity assumptions, the selectivity is:

→ $1/10 \times 1/100 = 0.001$

But since only Honda makes Accords the real selectivity is $1/100 = 0.01$



COST ESTIMATIONS

Our formulas are nice, but we assume that data values are uniformly distributed.

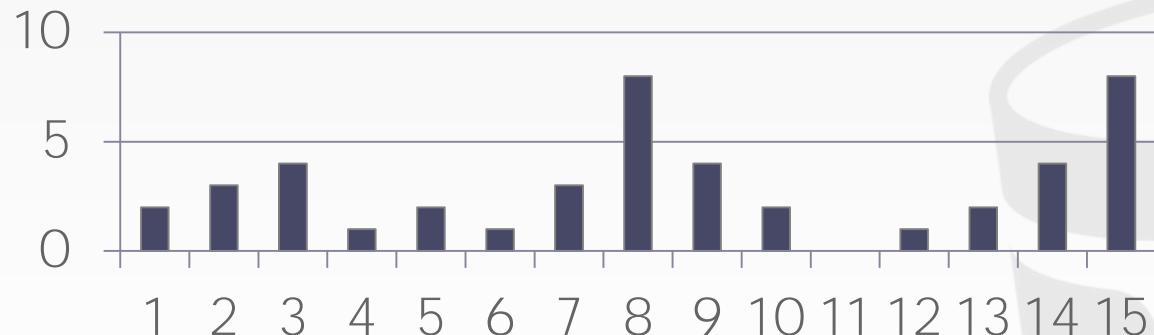
Uniform Approximation



COST ESTIMATIONS

Our formulas are nice, but we assume that data values are uniformly distributed.

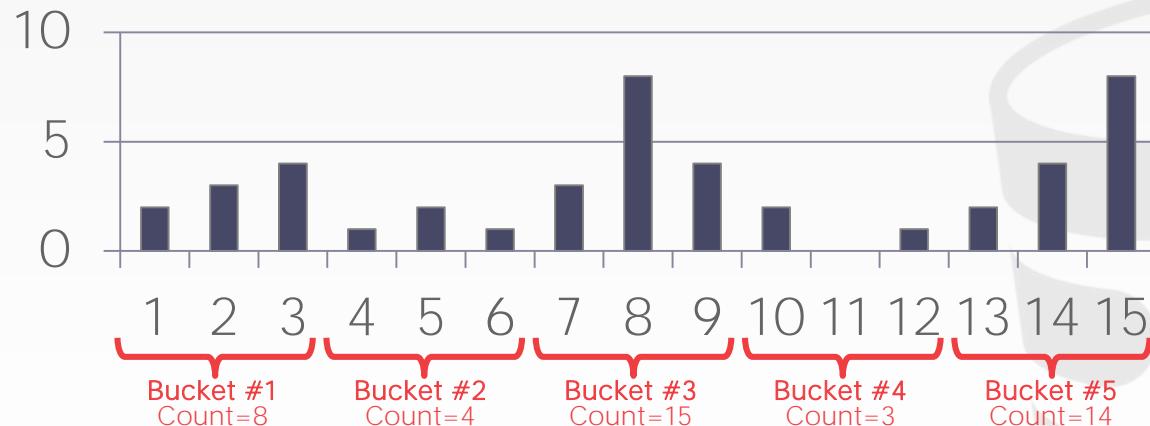
Non-Uniform Approximation



COST ESTIMATIONS

Our formulas are nice, but we assume that data values are uniformly distributed.

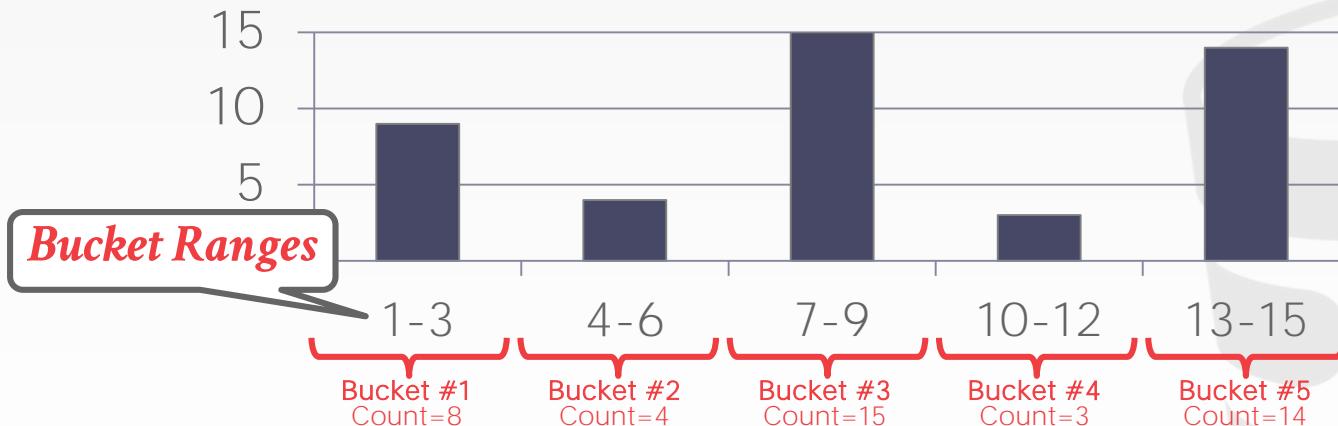
Non-Uniform Approximation



COST ESTIMATIONS

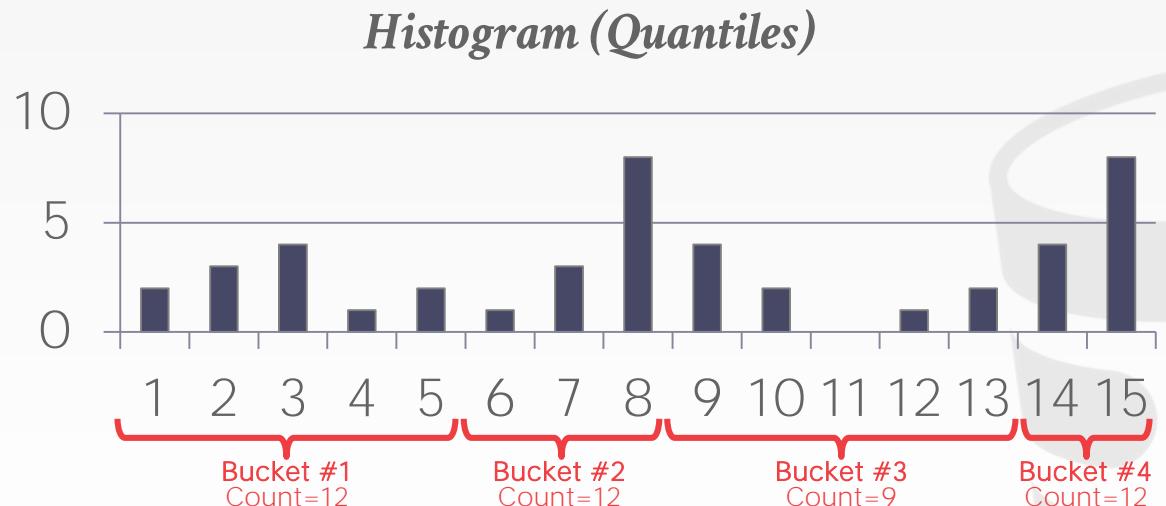
Our formulas are nice, but we assume that data values are uniformly distributed.

Non-Uniform Approximation



HISTOGRAMS WITH QUANTILES

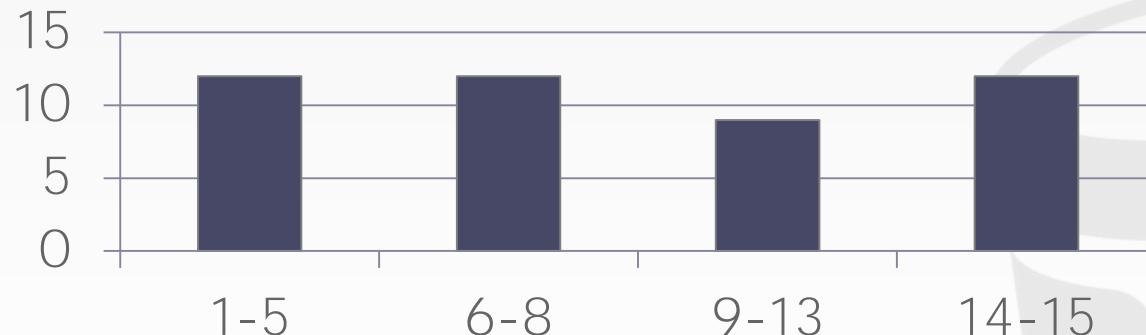
Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



HISTOGRAMS WITH QUANTILES

Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.

Histogram (Quantiles)



SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)  
FROM people  
WHERE age > 50
```

id	name	age	status
1001	Obama	58	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	25	Crunk
1005	Andy	38	Lit

⋮
1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

Table Sample

1001	Obama	58	Rested
1003	Tupac	25	Dead
1005	Andy	38	Lit

```
SELECT AVG(age)
  FROM people
WHERE age > 50
```

id	name	age	status
1001	Obama	58	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	25	Crunk
1005	Andy	38	Lit

⋮
1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	58	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	25	Crunk
1005	Andy	38	Lit



Table Sample

1001	Obama	58	Rested
1003	Tupac	25	Dead
1005	Andy	38	Lit

$\text{sel}(\text{age}>50) = 1/3$

⋮
1 billion tuples

OBSERVATION

Now that we can (roughly) estimate the selectivity of predicates, what can we actually do with them?

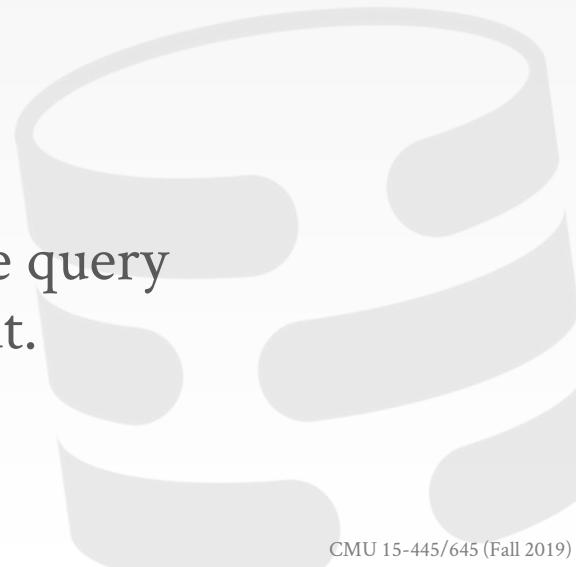


QUERY OPTIMIZATION

After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.

- Single relation.
- Multiple relations.
- Nested sub-queries.

It chooses the best plan it has seen for the query after exhausting all plans or some timeout.



SINGLE-RELATION QUERY PLANNING

Pick the best access method.

- Sequential Scan
- Binary Search (clustered indexes)
- Index Scan

Predicate evaluation ordering.

Simple heuristics are often good enough for this.

OLTP queries are especially easy...



OLTP QUERY PLANNING

Query planning for OLTP queries is easy because they are **sargable** (Search Argument Able).

- It is usually just picking the best index.
- Joins are almost always on foreign key relationships with a small cardinality.
- Can be implemented with simple heuristics.

```
CREATE TABLE people (
    id INT PRIMARY KEY,
    val INT NOT NULL,
    ...
);
```

```
SELECT * FROM people
WHERE id = 123;
```

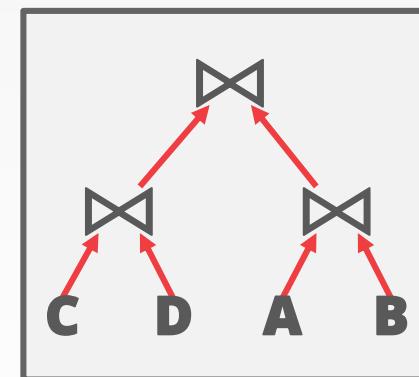
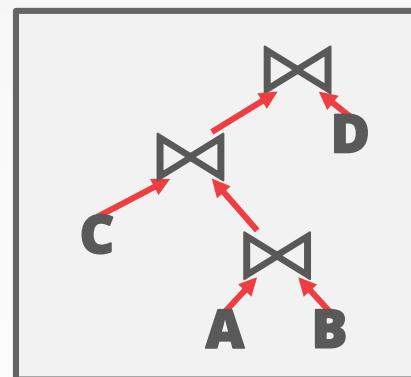
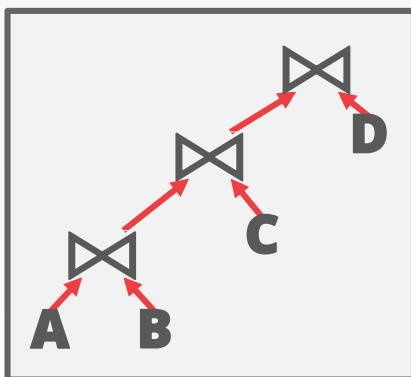
MULTI-RELATION QUERY PLANNING

As number of joins increases, number of alternative plans grows rapidly
→ We need to restrict search space.

Fundamental decision in **System R**: only left-deep join trees are considered.
→ Modern DBMSs do not always make this assumption anymore.

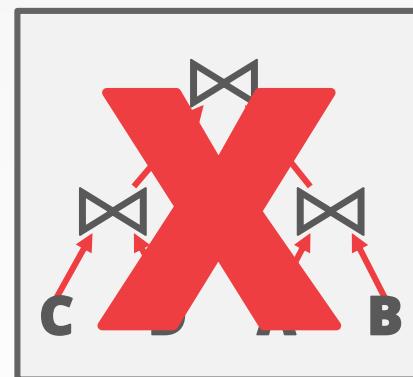
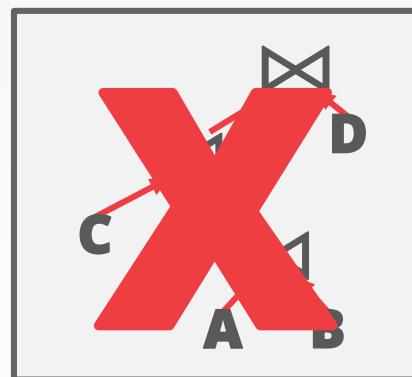
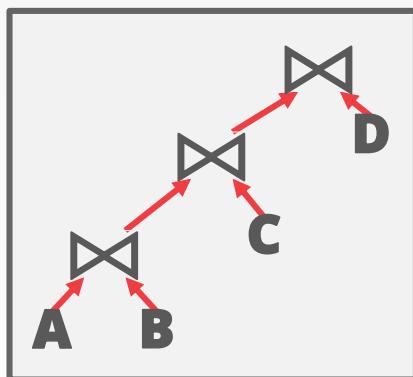
MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.



MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.



MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.

Allows for fully pipelined plans where intermediate results are not written to temp files.
→ Not all left-deep trees are fully pipelined.



MULTI-RELATION QUERY PLANNING

Enumerate the orderings

→ Example: Left-deep tree #1, Left-deep tree #2...

Enumerate the plans for each operator

→ Example: Hash, Sort-Merge, Nested Loop...

Enumerate the access paths for each table

→ Example: Index #1, Index #2, Seq Scan...

Use **dynamic programming** to reduce the number of cost estimations.



DYNAMIC PROGRAMMING

Hash Join

$R.a=S.a$

Cost:
300

$R \bowtie S$
T

SortMerge Join

$R.a=S.a$

Cost:
400

R
S
T

SortMerge Join

$T.b=S.b$

Cost:
280

$T \bowtie S$
R

Hash Join

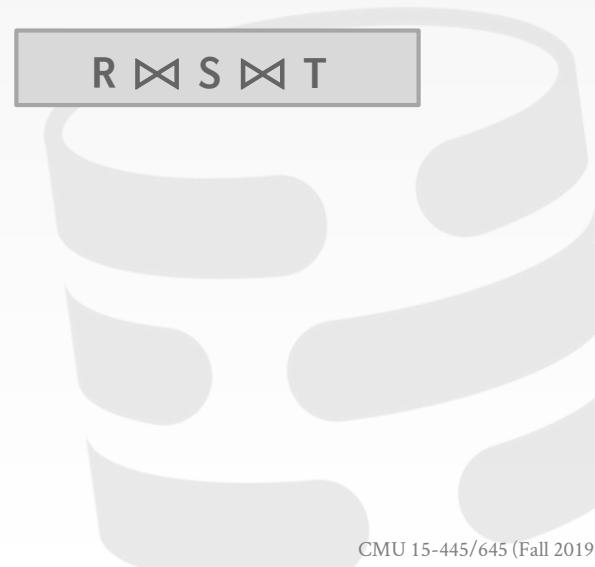
$T.b=S.b$

Cost:
200

⋮

```
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
```

$R \bowtie S \bowtie T$



DYNAMIC PROGRAMMING

Hash Join

$R.a=S.a$

Cost:
300

$R \bowtie S$
T

R
S
T

Hash Join

$T.b=S.b$

Cost:
200

$T \bowtie S$
R

⋮

**SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b**

$R \bowtie S \bowtie T$

DYNAMIC PROGRAMMING

Hash Join

R.a=S.a

Cost:
300

R \bowtie S
T

Hash Join

S.b=T.b

Cost:
380

**SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b**

R
S
T

SortMerge Join

S.b=T.b

Cost:
400

T \bowtie S
R

Hash Join

T.b=S.b

Cost:
200

SortMerge Join

S.a=R.a

Cost:
300

R \bowtie S \bowtie T

Hash Join

S.a=R.a

Cost:
450

DYNAMIC PROGRAMMING

Hash Join

R.a=S.a

Cost:
300

R \bowtie S
T

Hash Join

S.b=T.b

Cost:
380

**SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b**

R
S
T

Hash Join

T.b=S.b

Cost:
200

T \bowtie S
R

⋮
⋮

R \bowtie S \bowtie T

SortMerge Join

S.a=R.a
Cost:
300

DYNAMIC PROGRAMMING

R \bowtie S
T

**SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b**

R
S
T

Hash Join

T.b=S.b Cost: 200

SortMerge Join
S.a=R.a Cost: 300

T \bowtie S
R

⋮

R \bowtie S \bowtie T

CANDIDATE PLAN EXAMPLE

How to generate plans for search algorithm:

- Enumerate relation orderings
- Enumerate join algorithm choices
- Enumerate access method choices

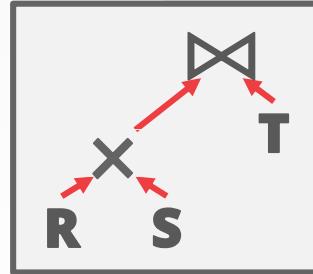
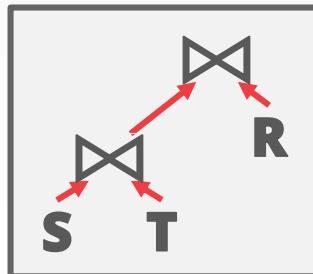
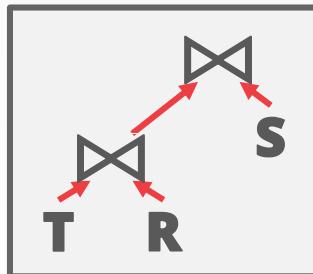
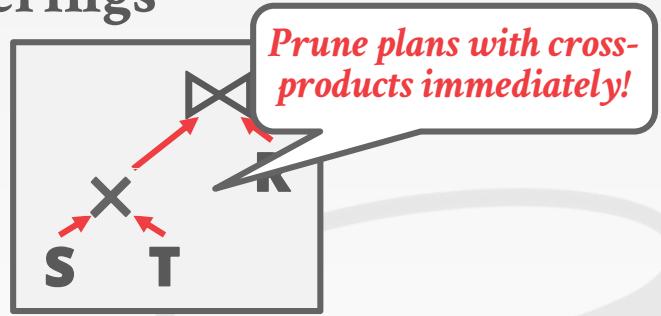
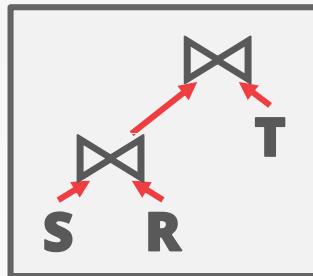
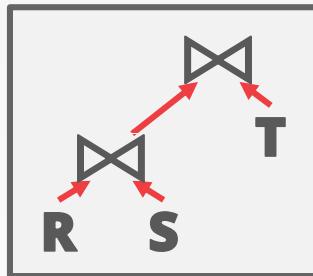
No real DBMSs does it this way.
It's actually more messy...

```
SELECT * FROM R, S, T  
WHERE R.a = S.a  
AND S.b = T.b
```



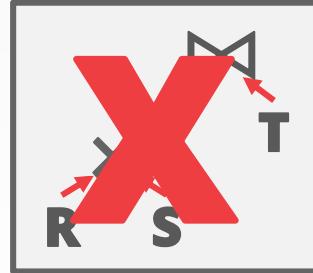
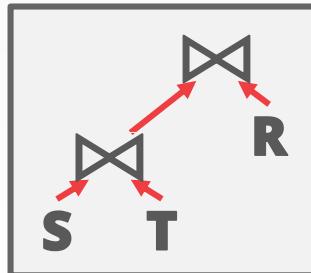
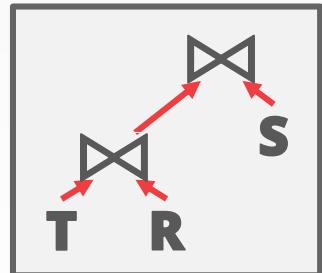
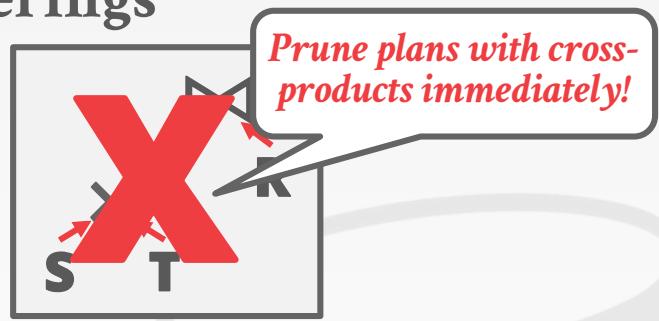
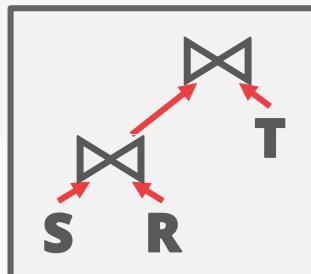
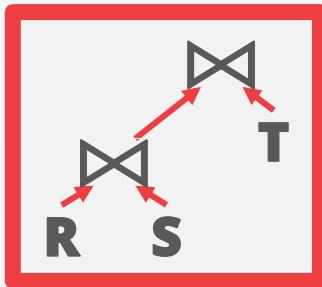
CANDIDATE PLANS

Step #1: Enumerate relation orderings



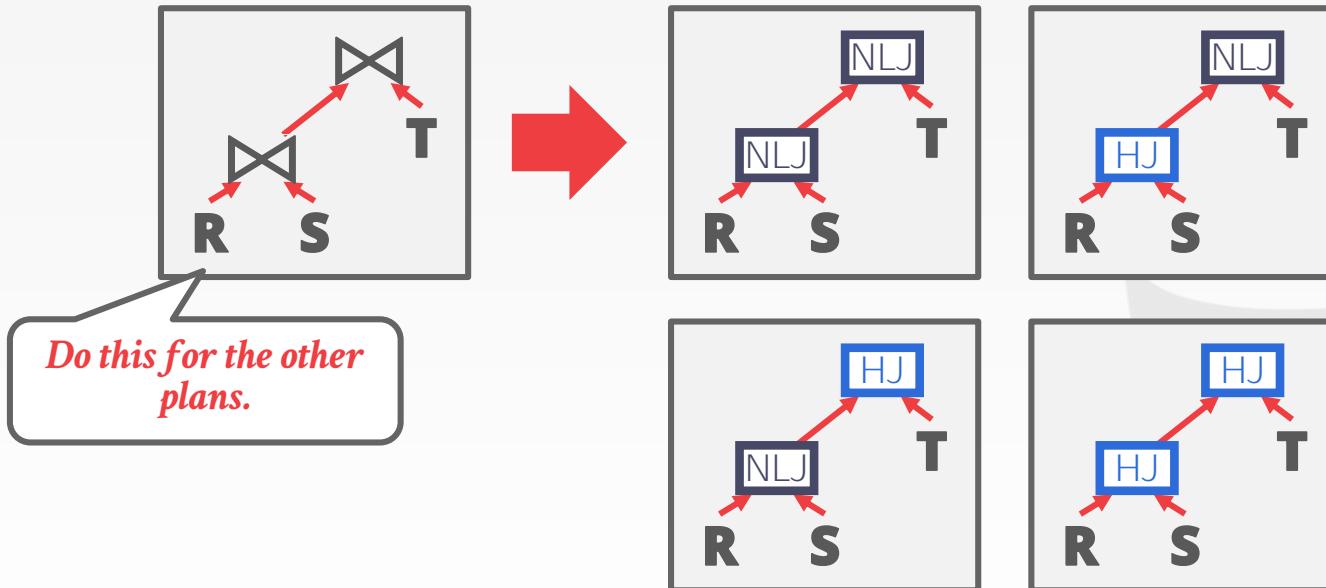
CANDIDATE PLANS

Step #1: Enumerate relation orderings



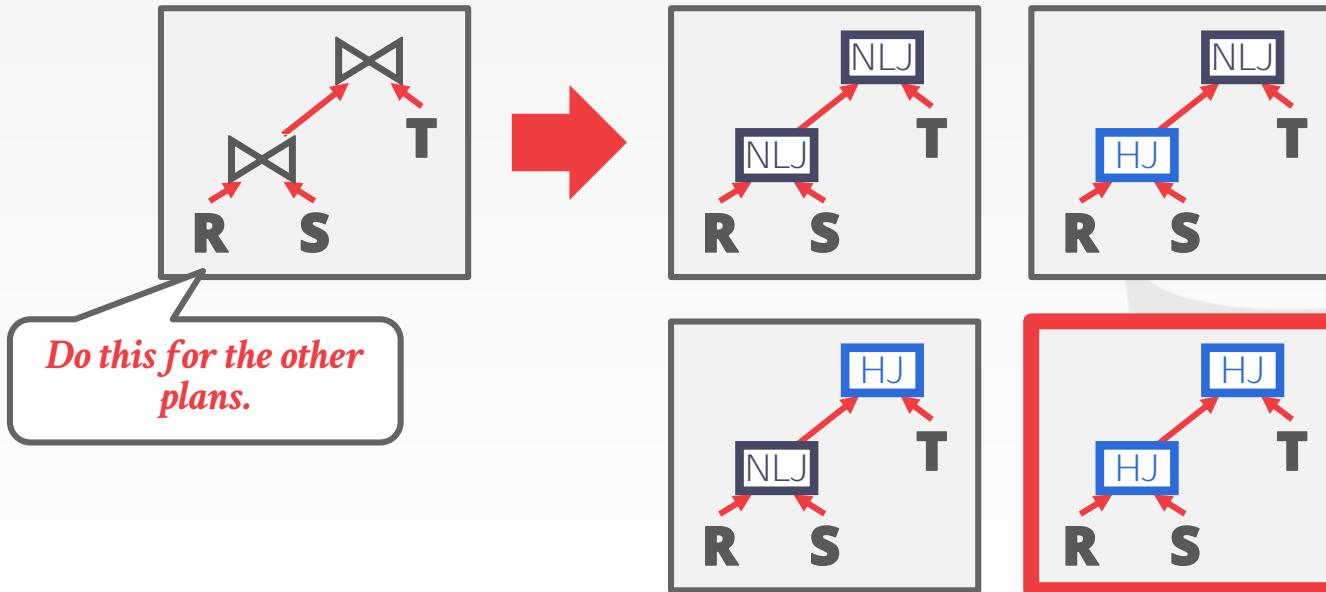
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices



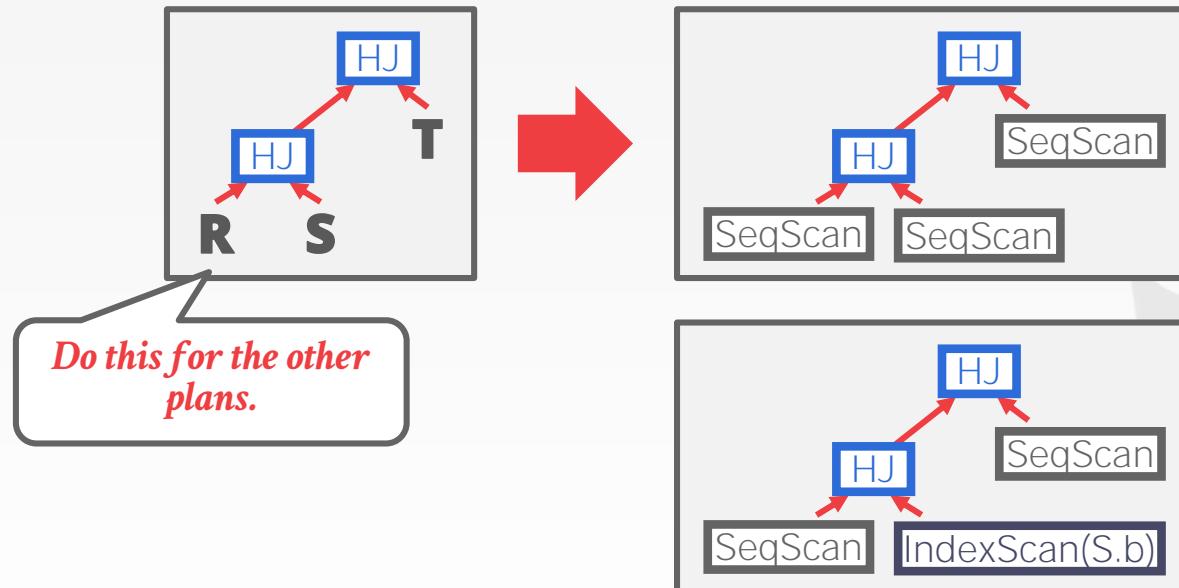
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices



CANDIDATE PLANS

Step #3: Enumerate access method choices



POSTGRES OPTIMIZER

Examines all types of join trees

→ Left-deep, Right-deep, bushy

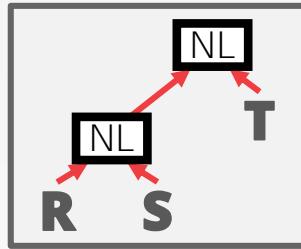
Two optimizer implementations:

→ Traditional Dynamic Programming Approach
→ Genetic Query Optimizer (GEQO)

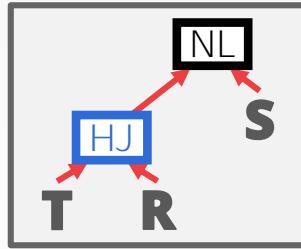
Postgres uses the traditional algorithm when # of tables in query is less than 12 and switches to GEQO when there are 12 or more.

POSTGRES OPTIMIZER

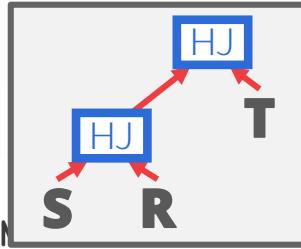
1st Generation



Cost:
300



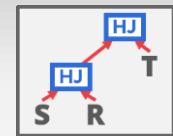
Cost:
200



Cost:
100

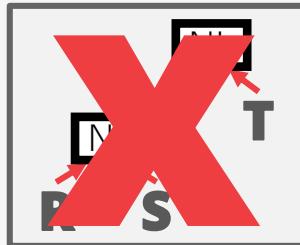


POSTGRES OPTIMIZER

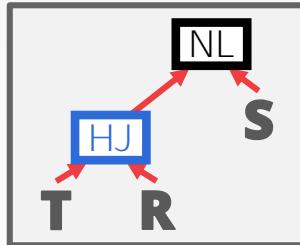


Best: 100

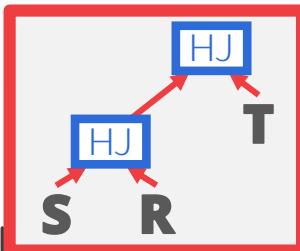
1st Generation



Cost:
300



Cost:
200

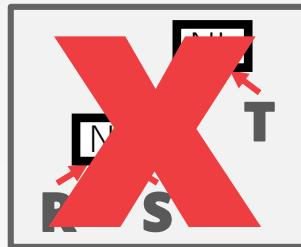


Cost:
100

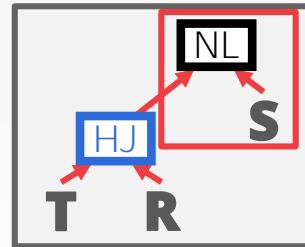


POSTGRES OPTIMIZER

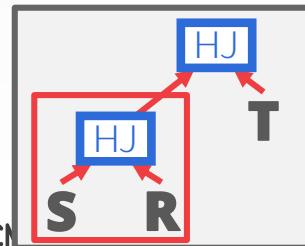
1st Generation



Cost: 300

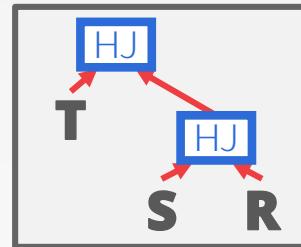


Cost: 200

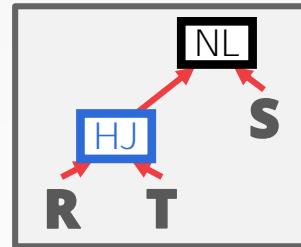


Cost: 100

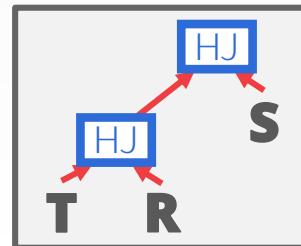
2nd Generation



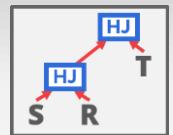
Cost: 80



Cost: 200



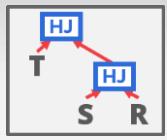
Cost: 110



Best: 100

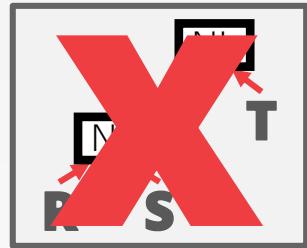


POSTGRES OPTIMIZER

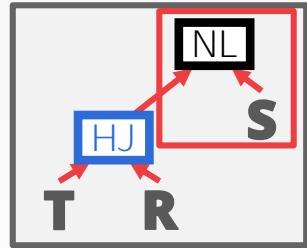


Best: 80

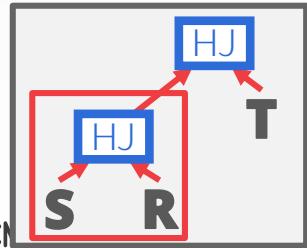
1st Generation



Cost: 300

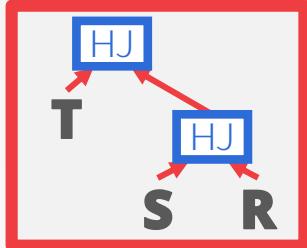


Cost: 200

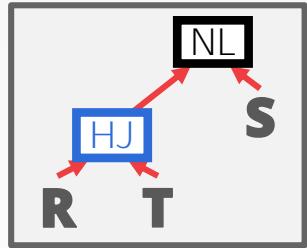


Cost: 100

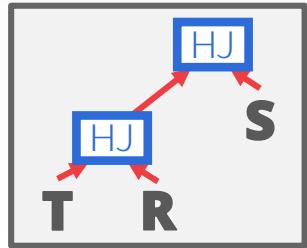
2nd Generation



Cost: 80



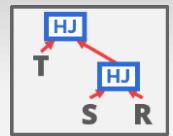
Cost: 200



Cost: 110

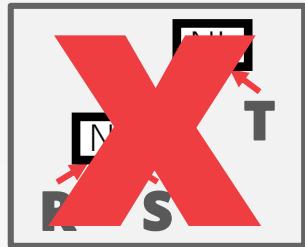


POSTGRES OPTIMIZER

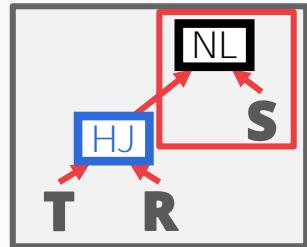


Best: 80

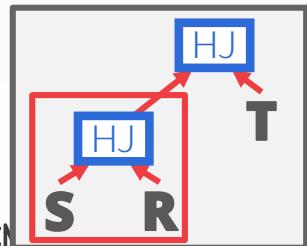
1st Generation



Cost: 300

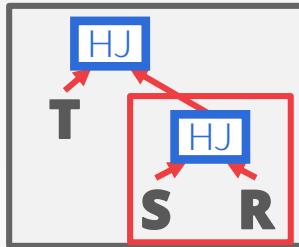


Cost: 200

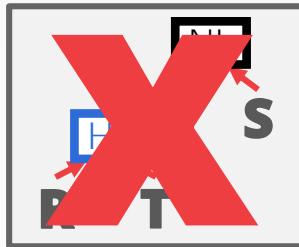


Cost: 100

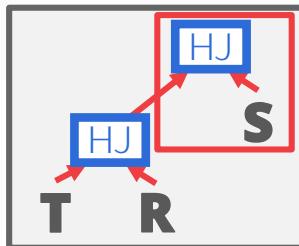
2nd Generation



Cost: 80



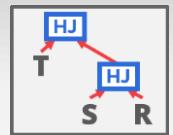
Cost: 200



Cost: 110

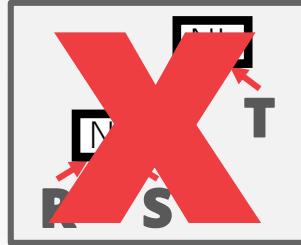


POSTGRES OPTIMIZER

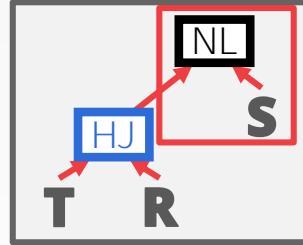


Best: 80

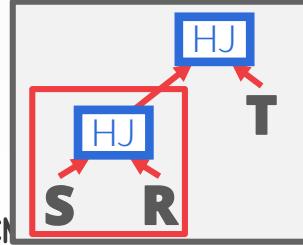
1st Generation



Cost: 300

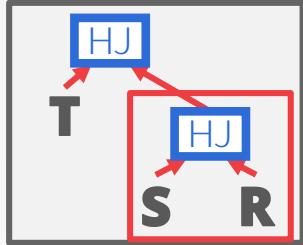


Cost: 200

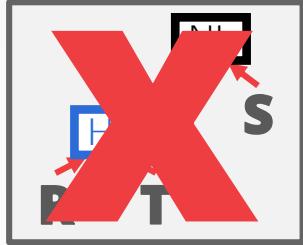


Cost: 100

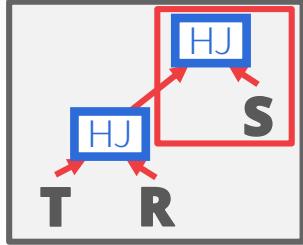
2nd Generation



Cost: 80

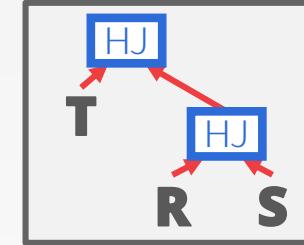


Cost: 200

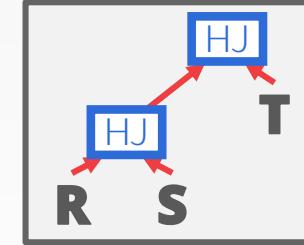


Cost: 110

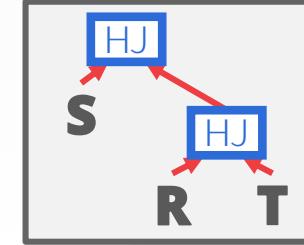
3rd Generation



Cost: 90



Cost: 160



Cost: 120

NESTED SUB-QUERIES

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:

- Rewrite to de-correlate and/or flatten them
- Decompose nested query and store result to temporary table

NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
WHERE EXISTS (
    SELECT * FROM reserves AS R
    WHERE S.sid = R.sid
    AND R.day = '2018-10-15'
)
```

NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
WHERE EXISTS (
    SELECT * FROM reserves AS R
    WHERE S.sid = R.sid
    AND R.day = '2018-10-15'
)
```



```
SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2018-10-15'
```

NESTED SUB-QUERIES: DECOMPOSE

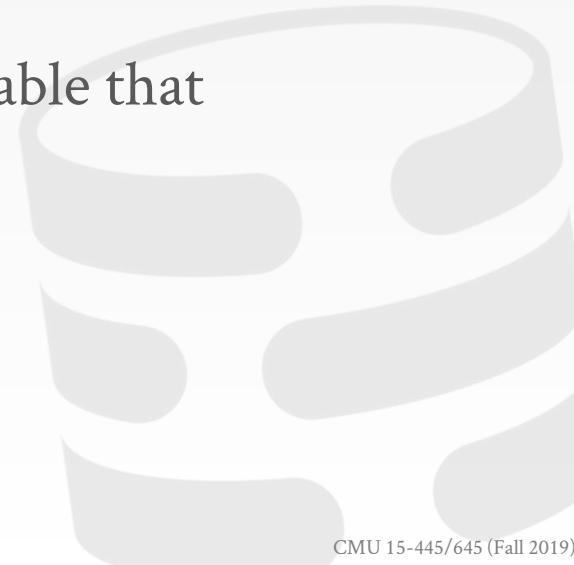
```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                    FROM sailors S2)
 GROUP BY S.sid
 HAVING COUNT(*) > 1
```

For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

DECOMPOSING QUERIES

For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.

Sub-queries are written to a temporary table that are discarded after the query finishes.



DECOMPOSING QUERIES

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                    FROM sailors S2)
 GROUP BY S.sid
 HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                    FROM sailors S2)
 GROUP BY S.sid
 HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ### ←
```

```
GROUP BY S.sid
HAVING COUNT(*) > 1
```

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ### ←
```

```
GROUP BY S.sid
HAVING COUNT(*) > 1
```

Outer Block

CONCLUSION

Filter early as possible.

Selectivity estimations

- Uniformity
- Independence
- Histograms
- Join selectivity

Dynamic programming for join orderings

Rewrite nested queries

Again, query optimization is hard...



EXTRA CREDIT

Each student can earn extra credit if they write a encyclopedia article about a DBMS.

→ Can be academic/commercial, active/historical.

Each article will use a standard taxonomy.

- For each feature category, you select pre-defined options for your DBMS.
- You will then need to provide a summary paragraph with citations for that category.

Database of Databases Browse Leaderboards Recent Accounts ▾

Database of Databases

Discover and learn about 657 database management systems

Search

[Browse](#) [Leaderboards](#)

Most Recent	Most Viewed	Most Edited
 Hyrise	 TiDB	 Derby
 TabDB	 LevelDB	 DynamoDB
 IoTDB	 RocksDB	 FaunaDB
 ThruDB	 NeDB	 ZODB
 KareDB	 LMDB	 Prometheus

Copyright © 2019 • Carnegie Mellon Database Group

Contact Github

R EDIT

credit if they write a DBMS.
active/historical.

and taxonomy.
select pre-defined options

summary paragraph with

Database of Databases Browse Leaderboards Recent

Database of Databases

Discover a database

Search

Start Year

End Year

Country

- Australia
- Austria
- Bangladesh

Show more

Compatible With

- Access
- Caché
- Cassandra

Show more

Embeds / Uses

- Berkeley DB
- BoltDB
- Cassandra

Show more

Derived From

- Accumulo
- Adaptive Server Enterprise
- Btrieve

Show more

Inspired By

- BigQuery
- C-Store
- Calvin

Show more

Operating System

- AIX
- All OS with Java VM
- Android

Show more

Programming Languages

- ActionScript
- Assembly
- Bash
- Rust

Show more

Project Types

Refine by

Search

Begin searching!

Search

Found 8 databases



IndraDB

Last updated July 18, 2019, 6:34 p.m.



Mentat

Last updated July 19, 2019, 12:11 a.m.



Sled

Last updated May 14, 2019, 10:43 p.m.



LlamaDB

Last updated May 16, 2018, 7:20 p.m.



PumpkinDB

Last updated Dec. 30, 2018, 10:04 a.m.



TerminusDB

Last updated Sept. 24, 2019, noon



LocustDB

Last updated July 18, 2019, 6:42 p.m.



TiKV

Last updated July 18, 2019, 5:59 p.m.

Copyright © 2019 • Carnegie Mellon Database Group

write a
ed options
graph with

Database of Databases Browse Leaderboards Recent

Database of Databases

Discover a database

Search

Most Recent

-  Hyrise
-  TDB
-  IoTDB
-  ThruDB
-  KareDB

Copyright © 2019 • Carnegie Mellon Database Group

Refine by

Start Year

End Year

Begin searching!

Country

- Australia
- Austria
- Bangladesh

Show more

Compatible With

- Access
- Caché
- Cassandra

Show more

Embeds / Uses

- Berkeley DB
- BoltDB
- Cassandra

Show more

Derived From

- Accumulo
- Adaptive Server Enterprise
- Btrieve

Show more

Inspired By

- BigQuery
- C-Store
- Calvin

Show more

Operating System

- AIX
- All OS with Java VM
- Android

Show more

Programming Languages

- ActionScript
- Assembly
- Bash
- Rust

Show more

Project Types

Database of Databases Browse Leaderboards Recent Revision List

Search

PumpkinDB

PumpkinDB is a low-level event sourcing database engine that is ACID-compliant. It is a database engine that could be used to build different types of event sourcing systems such as embedded and client-server ones.

PumpkinDB is designed to be immutable, the reason behind this is that overwriting data could be unsafe, valuable history of data will be erased. As the cost of storage dropping, a more effective way of managing data is to enforcing immutability of key's value. And writing to the database is close to functional, such that writing side doesn't have to wait for shared resources, the read side will figure out the correct result.

In order to have control over querying costs, it provides an embedded executable imperative language, PumpkinScript, which is a low-level untyped language inspired by MUMPS.

PumpkinDB does not have custom protocols for communication, instead, it has a pipeline to a script executor. When the applications need to communicate with PumpkinDB, small PumpkinScript programs are sent through a network interface in order to do that.

History

PumpkinDB is a descendant of a event capture and querying framework ES4j. The difference from ES4j is that PumpkinDB has a HLC timestamp, a UUID, complies with the ELF format, and 2017.

Concurrency Control

Not Supported

Data Model

KeyValue

It supports binary keys and values, which enables the use of any encoding such as XML, JSON and Protobuf.

Website <http://pumpkindb.org/>

Source Code <https://github.com/PumpkinDB/PumpkinDB>

Tech Docs <http://pumpkindb.org/doc/>

Developer Yurii Rashkovskii

Country of Origin CA

Start Year 2017

Project Type Open Source

Written in Rust

Operating Systems Linux, Windows

Licenses Mozilla Public License



DBDB.IO

All the articles will be hosted on dbdb.io
→ I will post registration details on Piazza.

I will post a sign-up sheet for you to pick what DBMS you want to write about.

- If you choose a widely known DBMS, then the article will need to be comprehensive.
- If you choose an obscure DBMS, then you will have to do the best you can to find information.

All the articles will be
→ I will post registration

I will post a sign-up sheet
DBMS you want to work with
→ If you choose a widely used DBMS you need to be comprehensive
→ If you choose an obscure DBMS the best you can to find

The screenshot shows a web-based form titled "Edit Database System". The form is for creating or updating a database entry. Key fields include:

- Name:** PumpkinDB
- Logo:** A file named "logos/pumpkindb.png" is currently selected.
- Url:** http://pumpkindb.org/
- Developer:** Yurii Rashkovskii
- Source Code URL:** https://github.com/PumpkinDB/PumpkinDB
- Tech docs:** http://pumpkindb.org/doc/
- Wikipedia URL:** Wikipedia URL
- Project Type:** Open Source (selected)
- Countries of Origin:** Canada
- Start year:** 2017
- End year:** End year
- Start year citations:** Separate the urls with commas
- End year citations:** Separate the urls with commas
- Former names:** Former names
- Acquired by:** Acquired by
- Acquired by citations:** Separate the urls with commas

Buttons on the right side of the form include "Save" and "Cancel". A "Revision Comment" section is also present.



HOW TO DECIDE

Pick a DBMS based on whatever criteria you want:

- Country of Origin
- Popularity
- Programming Language
- Single-Node vs. Embedded vs. Distributed
- Disk vs. Memory
- Row Store vs. Column Store
- Open-Source vs. Proprietary





PLAGIARISM WARNING



This article must be your own writing with your own images. You may **not** copy text/images directly from papers or other sources that you find on the web.

Plagiarism will **not** be tolerated.

See [CMU's Policy on Academic Integrity](#) for additional information.



NEXT CLASS

Transactions!

→ aka the second hardest part about database systems



16

Concurrency Control Theory



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #3 is due Sun Nov 17th @ 11:59pm.

Homework #4 will be released next week.
It is due Wed Nov 13th @ 11:59pm.



COURSE STATUS

A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.

Query Planning

Operator Execution

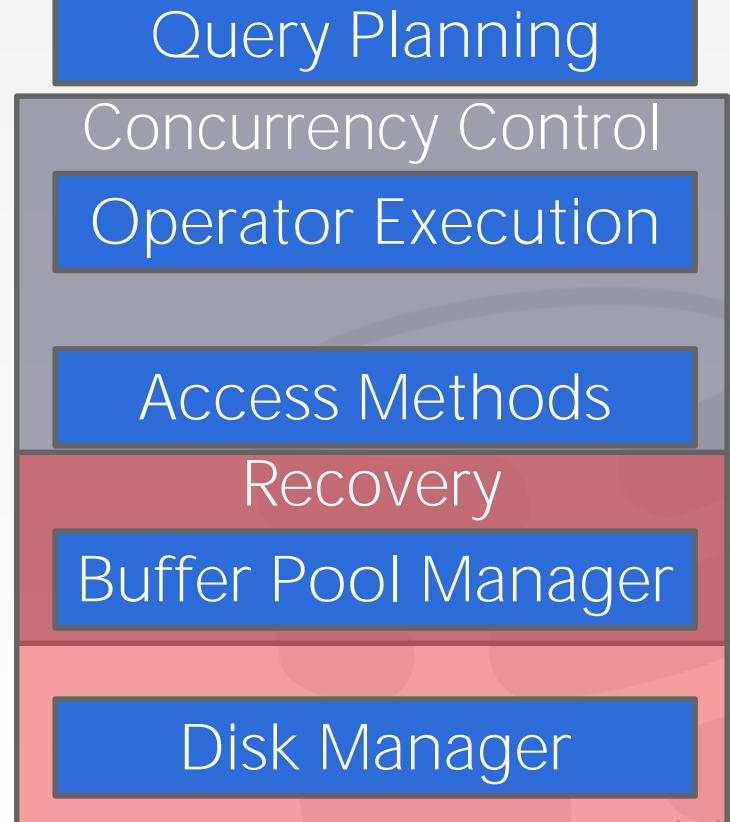
Access Methods

Buffer Pool Manager

Disk Manager

COURSE STATUS

A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.



MOTIVATION

We both change the same record in a table at the same time.

How to avoid race condition?



Lost Updates
Concurrency Control

You transfer \$100 between bank accounts but there is a power failure.
What is the correct database state?



Durability
Recovery

CONCURRENCY CONTROL & RECOVERY

Valuable properties of DBMSs.

Based on concept of transactions with **ACID** properties.

Let's talk about transactions...



TRANSACTIONS

A **transaction** is the execution of a sequence of one or more operations (e.g., SQL queries) on a database to perform some higher-level function.

It is the basic unit of change in a DBMS:
→ Partial transactions are not allowed!



TRANSACTION EXAMPLE

Move \$100 from Andy' bank account to his promotor's account.

Transaction:

- Check whether Andy has \$100.
- Deduct \$100 from his account.
- Add \$100 to his promotor account.



STRAWMAN SYSTEM

Execute each txn one-by-one (i.e., serial order) as they arrive at the DBMS.

- One and only one txn can be running at the same time in the DBMS.

Before a txn starts, copy the entire database to a new file and make all changes to that file.

- If the txn completes successfully, overwrite the original file with the new one.
- If the txn fails, just remove the dirty copy.

PROBLEM STATEMENT

A (potentially) better approach is to allow concurrent execution of independent transactions.

Why do we want that?

- Better utilization/throughput
- Increased response times to users.

But we also would like:

- Correctness
- Fairness



TRANSACTIONS

Hard to ensure correctness...

- What happens if Andy only has \$100 and tries to pay off two promoters at the same time?

Hard to execute quickly...

- What happens if Andy tries to pay off his gambling debts at the exact same time?



PROBLEM STATEMENT

Arbitrary interleaving of operations can lead to:

- Temporary Inconsistency (ok, unavoidable)
- Permanent Inconsistency (bad!)

We need formal correctness criteria to determine whether an interleaving is valid.



DEFINITIONS

A txn may carry out many operations on the data retrieved from the database

However, the DBMS is only concerned about what data is read/written from/to the database.
→ Changes to the "outside world" are beyond the scope of the DBMS.

FORMAL DEFINITIONS

Database: A fixed set of named data objects (e.g.,
A, B, C, ...).

→ We do not need to define what these objects are now.

Transaction: A sequence of read and write
operations (**R(A), W(B), ...**)

→ DBMS's abstract view of a user program

TRANSACTIONS IN SQL

A new txn starts with the **BEGIN** command.

The txn stops with either **COMMIT** or **ABORT**:

- If commit, the DBMS either saves all the txn's changes or aborts it.
- If abort, all changes are undone so that it's like as if the txn never executed at all.

Abort can be either self-inflicted or caused by the DBMS.

CORRECTNESS CRITERIA: ACID

Atomicity: All actions in the txn happen, or none happen.

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one txn is isolated from that of other txns.

Durability: If a txn commits, its effects persist.

CORRECTNESS CRITERIA: ACID

Atomicity: “all or nothing”

Consistency: “it looks correct to me”

Isolation: “as if alone”

Durability: “survive failures”



TODAY'S AGENDA

Atomicity

Consistency

Isolation

Durability



ATOMICITY OF TRANSACTIONS

Two possible outcomes of executing a txn:

- Commit after completing all its actions.
- Abort (or be aborted by the DBMS) after executing some actions.

DBMS guarantees that txns are **atomic**.

- From user's point of view: txn always either executes all its actions, or executes no actions at all.



ATOMICITY OF TRANSACTIONS

Scenario #1:

- We take \$100 out of Andy's account but then the DBMS aborts the txn before we transfer it.

Scenario #2:

- We take \$100 out of Andy's account but then there is a power failure before we transfer it.

What should be the correct state of Andy's account after both txns abort?

MECHANISMS FOR ENSURING ATOMICITY

Approach #1: Logging

- DBMS logs all actions so that it can undo the actions of aborted transactions.
- Maintain undo records both in memory and on disk.
- Think of this like the black box in airplanes...

Logging is used by almost every DBMS.

- Audit Trail
- Efficiency Reasons



MECHANISMS FOR ENSURING ATOMICITY

Approach #2: Shadow Paging

- DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.
- Originally from System R.

Few systems do this:

- CouchDB
- LMDB (OpenLDAP)



MECHANISMS FOR ENSURING ATOMICITY

Approach #2: Shadow Paging

- DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.
- Originally from System R.

Few systems do this:

- CouchDB
- LMDB (OpenLDAP)



CONSISTENCY

The "world" represented by the database is logically correct. All questions asked about the data are given logically correct answers.

Database Consistency
Transaction Consistency



DATABASE CONSISTENCY

The database accurately models the real world and follows integrity constraints.

Transactions in the future see the effects of transactions committed in the past inside of the database.



TRANSACTION CONSISTENCY

If the database is consistent before the transaction starts (running alone), it will also be consistent after.

Transaction consistency is the application's responsibility.

→ We won't discuss this further...



ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it was running by itself.

→ Easier programming model to reason about.

But the DBMS achieves concurrency by interleaving the actions (reads/writes of DB objects) of txns.

We need a way to interleave txns but still make it appear as if they ran one-at-a-time.

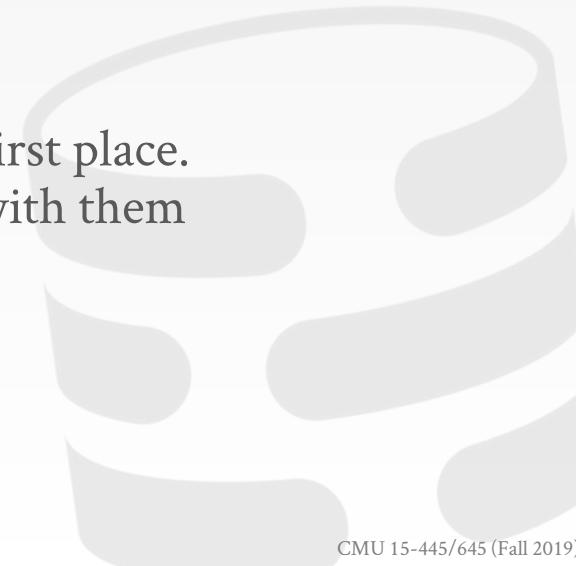


MECHANISMS FOR ENSURING ISOLATION

A **concurrency control** protocol is how the DBMS decides the proper interleaving of operations from multiple transactions.

Two categories of protocols:

- **Pessimistic**: Don't let problems arise in the first place.
- **Optimistic**: Assume conflicts are rare, deal with them after they happen.



EXAMPLE

Assume at first **A** and **B** each have \$1000.

T₁ transfers \$100 from **A**'s account to **B**'s

T₂ credits both accounts with 6% interest.

T₁

```
BEGIN  
A=A-100  
B=B+100  
COMMIT
```

T₂

```
BEGIN  
A=A*1.06  
B=B*1.06  
COMMIT
```

EXAMPLE

Assume at first **A** and **B** each have \$1000.

*What are the possible outcomes of running **T₁** and **T₂**?*

T₁

BEGIN

A=A-100

B=B+100

COMMIT

T₂

BEGIN

A=A*1.06

B=B*1.06

COMMIT



EXAMPLE

Assume at first **A** and **B** each have \$1000.

What are the possible outcomes of running T_1 and T_2 ?

Many! But **A+B** should be:

$$\rightarrow \$2000 * 1.06 = \$2120$$

There is no guarantee that T_1 will execute before T_2 or vice-versa, if both are submitted together.
But the net effect must be equivalent to these two transactions running serially in some order.

EXAMPLE

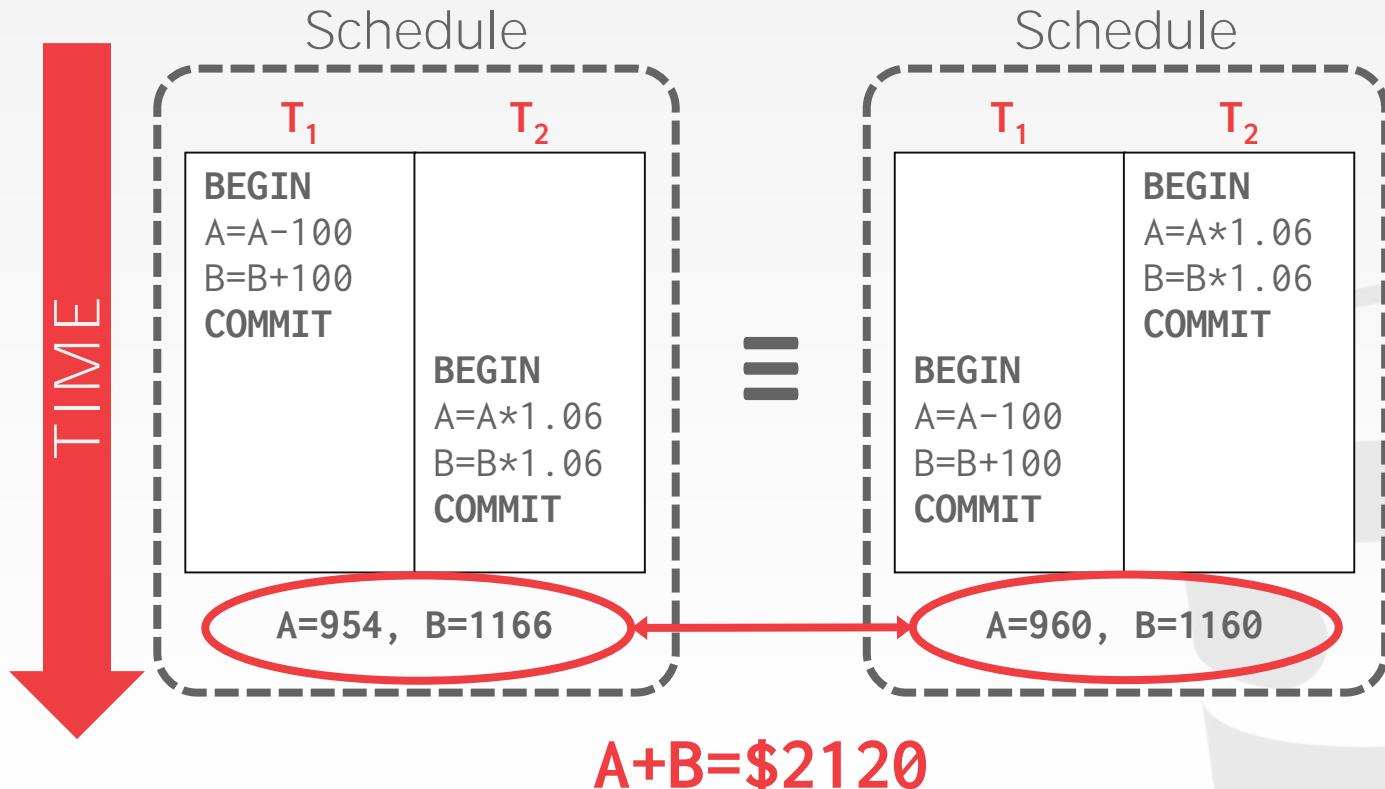
Legal outcomes:

- A=954, B=1166 → A+B=\$2120
- A=960, B=1160 → A+B=\$2120

The outcome depends on whether T_1 executes before T_2 or vice versa.



SERIAL EXECUTION EXAMPLE



INTERLEAVING TRANSACTIONS

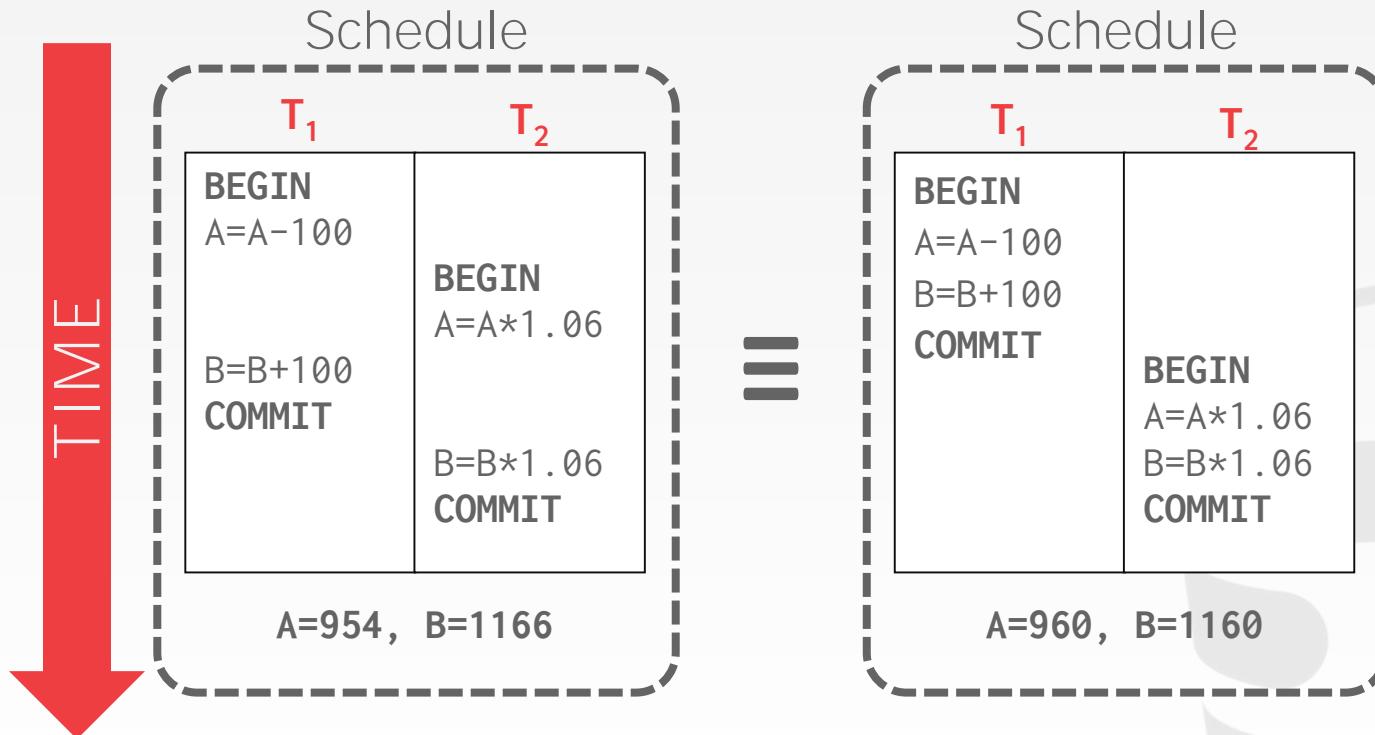
We interleave txns to maximize concurrency.

- Slow disk/network I/O.
- Multi-core CPUs.

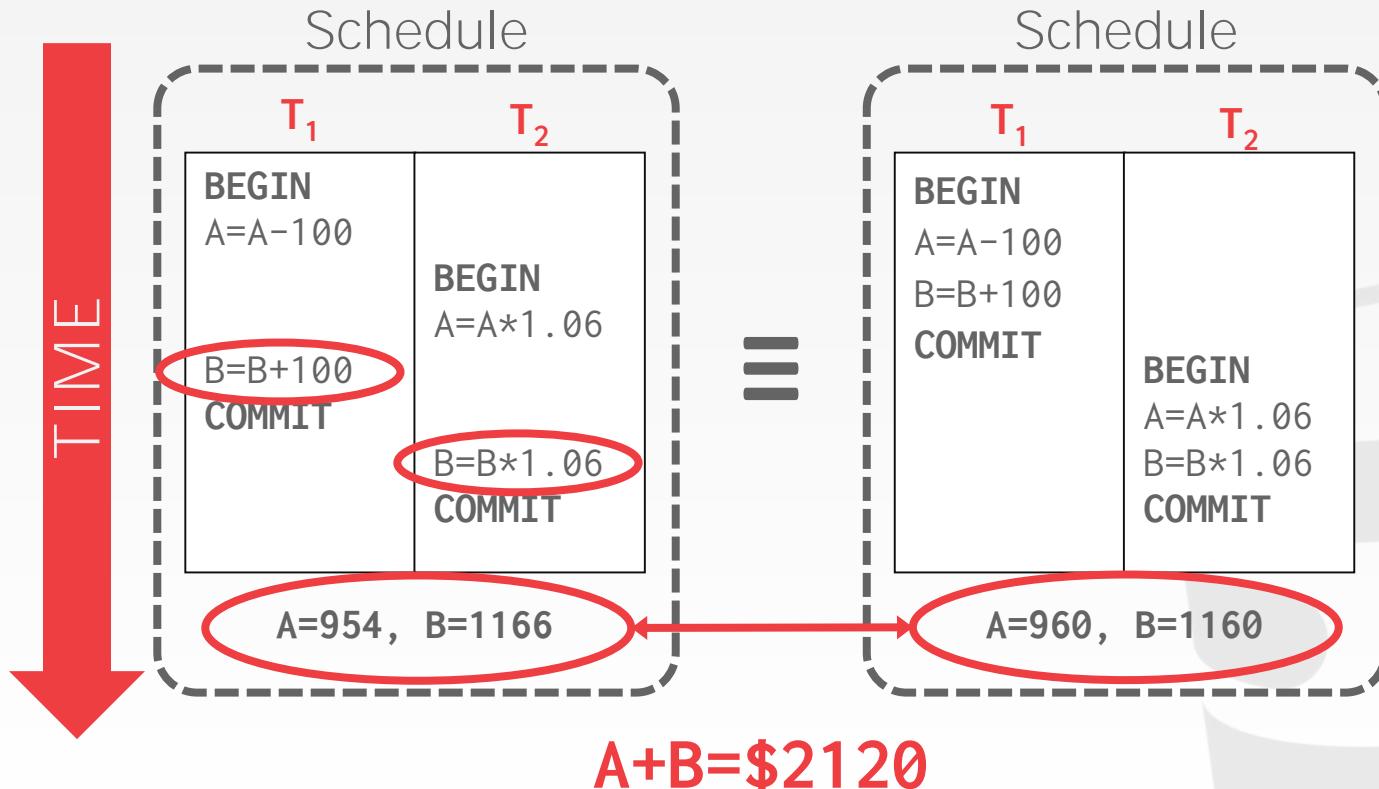
When one txn stalls because of a resource (e.g., page fault), another txn can continue executing and make forward progress.



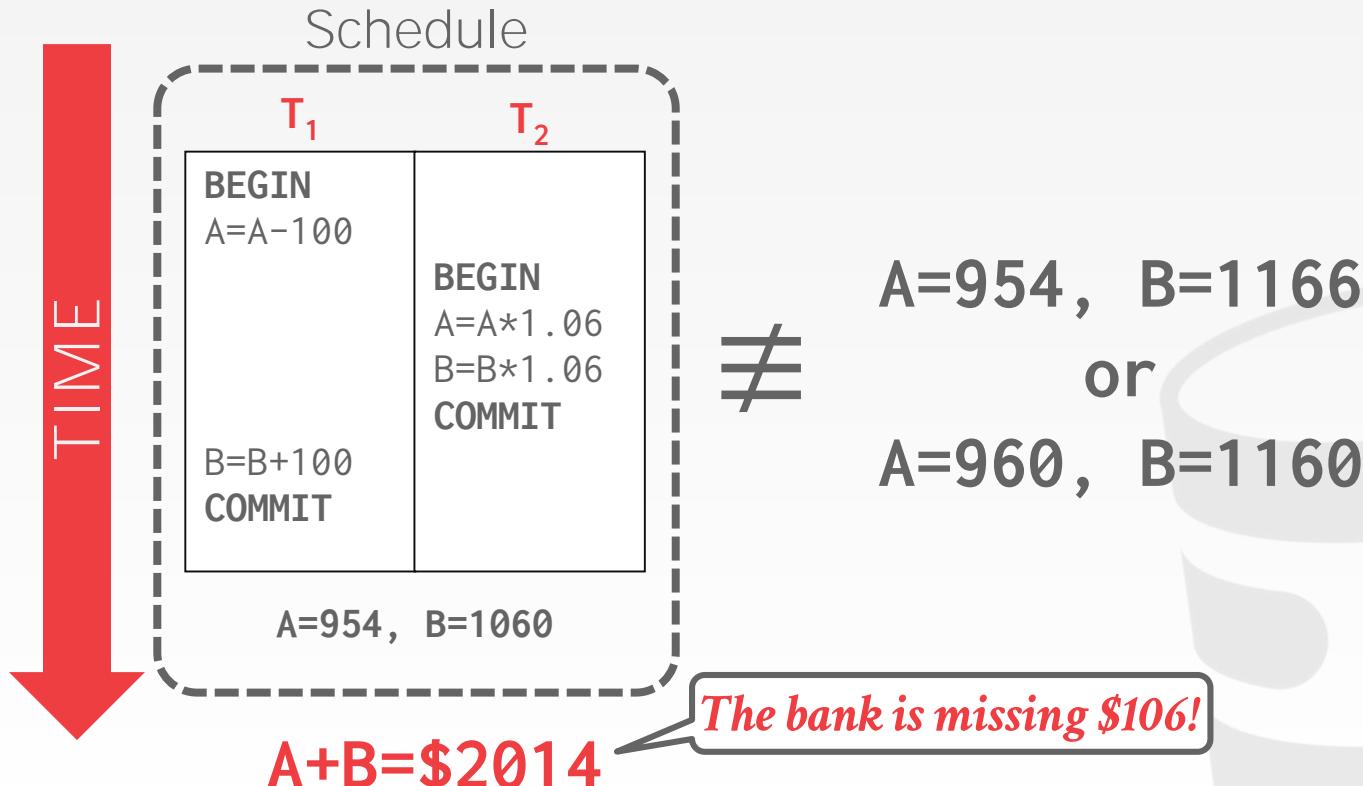
INTERLEAVING EXAMPLE (GOOD)



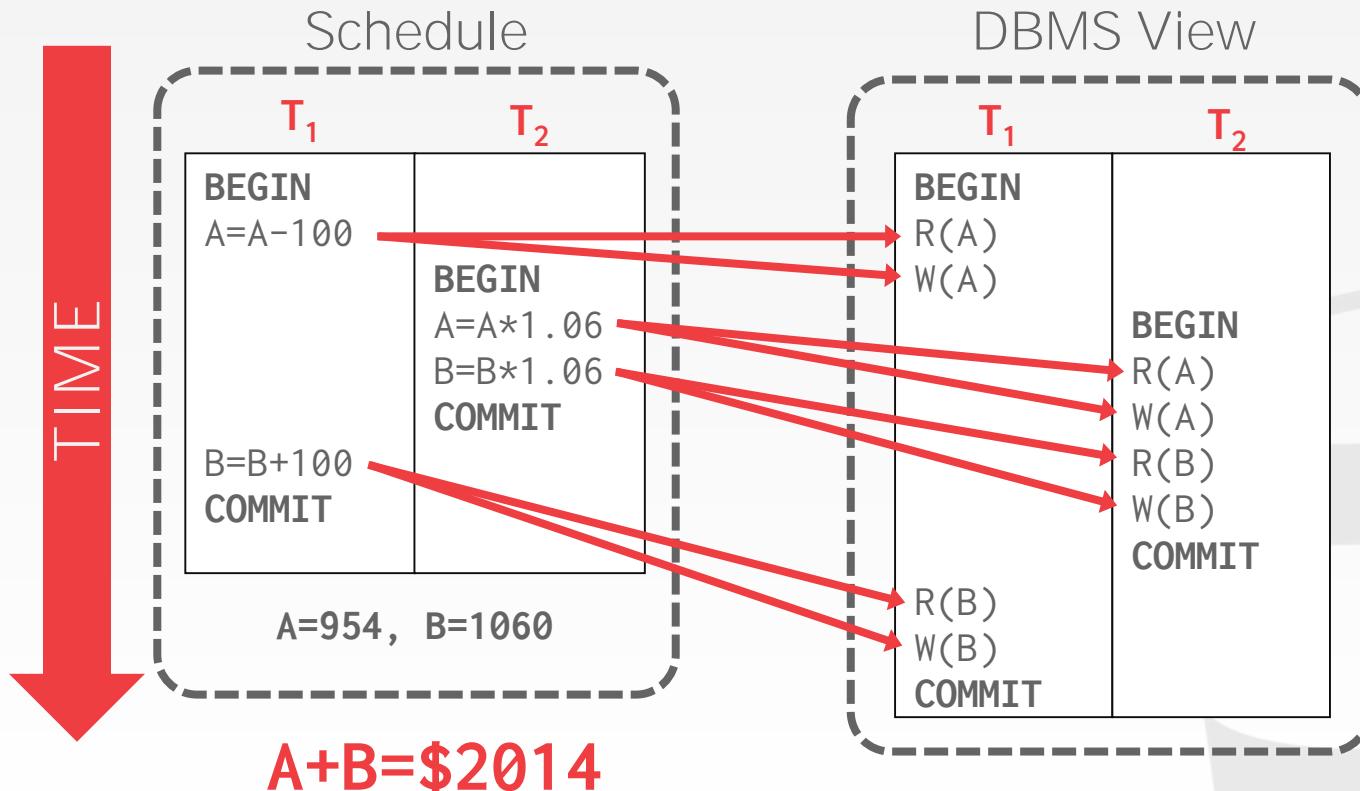
INTERLEAVING EXAMPLE (GOOD)



INTERLEAVING EXAMPLE (BAD)



INTERLEAVING EXAMPLE (BAD)



CORRECTNESS

How do we judge whether a schedule is correct?

If the schedule is equivalent to some serial execution.



FORMAL PROPERTIES OF SCHEDULES

Serial Schedule

- A schedule that does not interleave the actions of different transactions.

Equivalent Schedules

- For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- Doesn't matter what the arithmetic operations are!



FORMAL PROPERTIES OF SCHEDULES

Serializable Schedule

→ A schedule that is equivalent to some serial execution of the transactions.

If each transaction preserves consistency, every serializable schedule preserves consistency.



FORMAL PROPERTIES OF SCHEDULES

Serializability is a less intuitive notion of correctness compared to txn initiation time or commit order, but it provides the DBMS with additional flexibility in scheduling operations.

More flexibility means better parallelism.

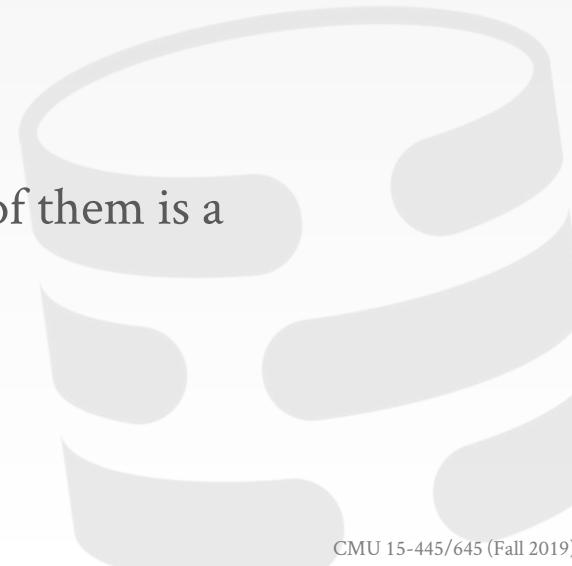


CONFICTING OPERATIONS

We need a formal notion of equivalence that can be implemented efficiently based on the notion of "conflicting" operations

Two operations **conflict** if:

- They are by different transactions,
- They are on the same object and at least one of them is a write.



INTERLEAVED EXECUTION ANOMALIES

Read-Write Conflicts (**R-W**)

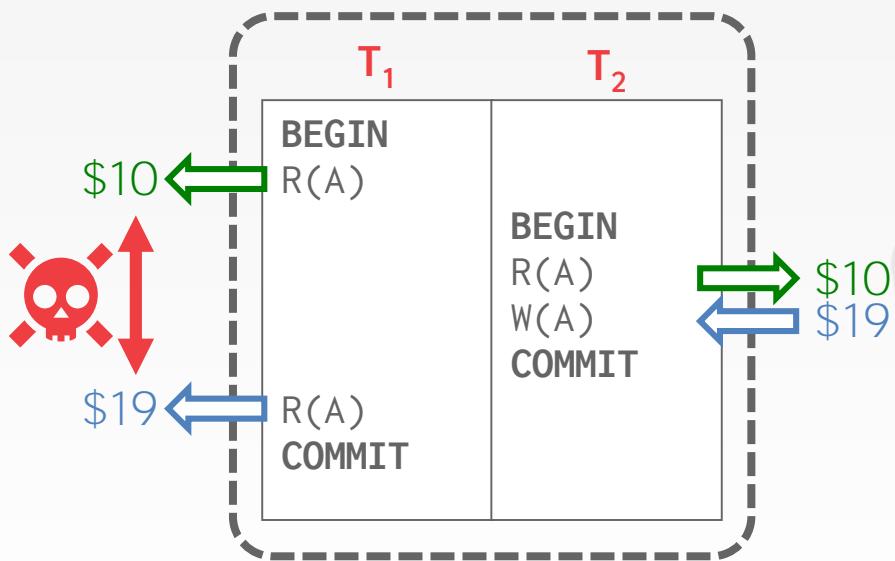
Write-Read Conflicts (**W-R**)

Write-Write Conflicts (**W-W**)



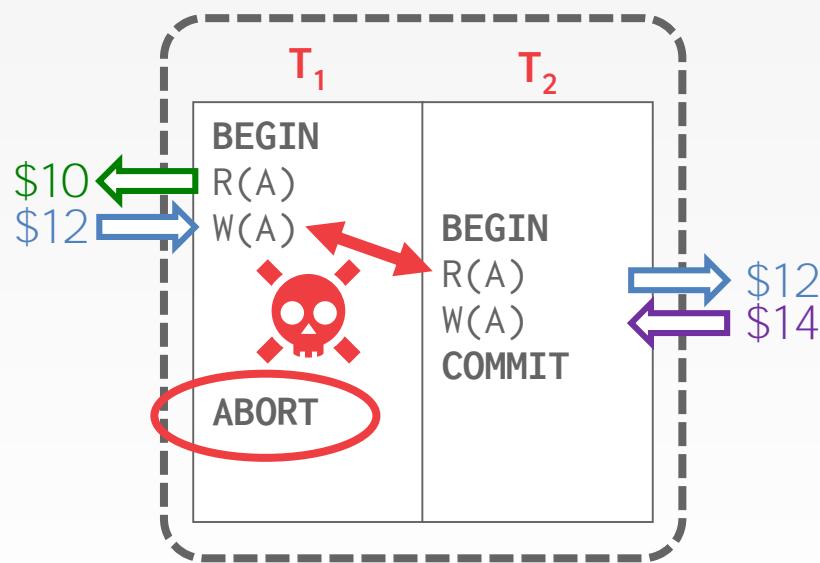
READ-WRITE CONFLICTS

Unrepeatable Reads



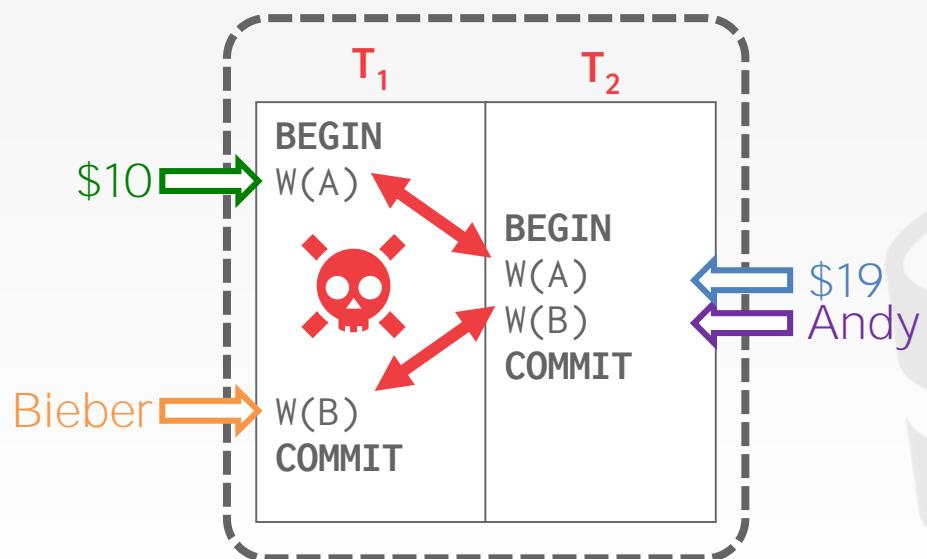
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")



WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data



FORMAL PROPERTIES OF SCHEDULES

Given these conflicts, we now can understand what it means for a schedule to be serializable.

- This is to check whether schedules are correct.
- This is not how to generate a correct schedule.

There are different levels of serializability:

- Conflict Serializability
- View Serializability

No DBMS can do this.

Most DBMSs try to support this.



CONFLICT SERIALIZABLE SCHEDULES

Two schedules are **conflict equivalent** iff:

- They involve the same actions of the same transactions,
and
- Every pair of conflicting actions is ordered the same way.

Schedule **S** is **conflict serializable** if:

- **S** is conflict equivalent to some serial schedule.



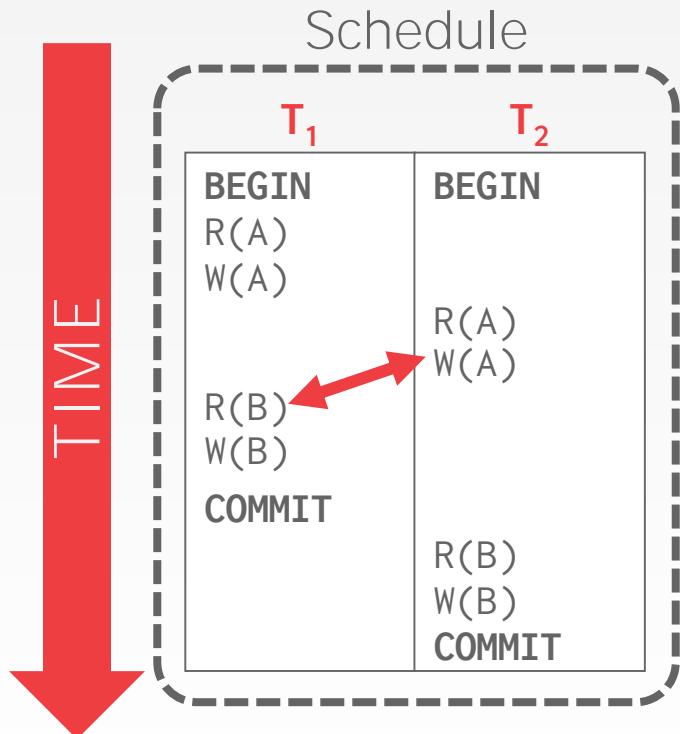


CONFLICT SERIALIZABILITY INTUITION

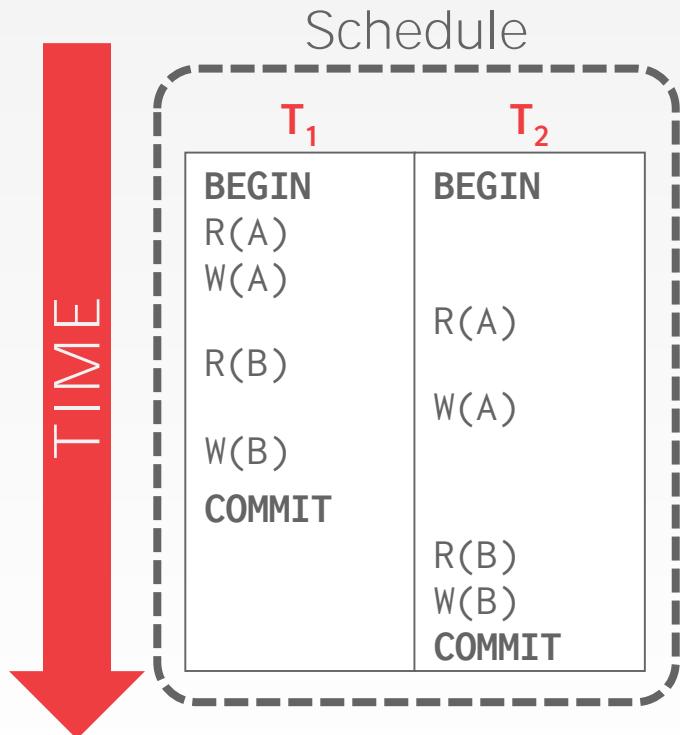
Schedule **S** is conflict serializable if you are able to transform **S** into a serial schedule by swapping consecutive non-conflicting operations of different transactions.



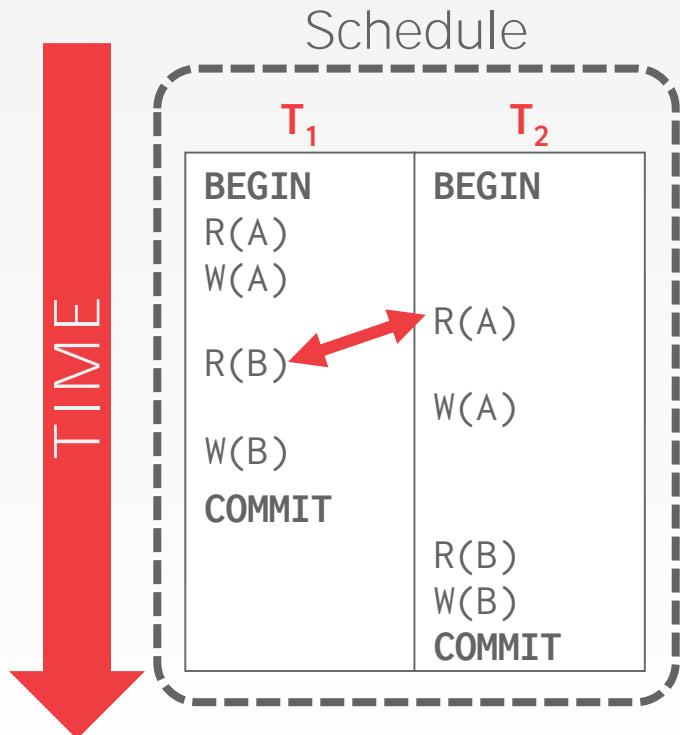
CONFLICT SERIALIZABILITY INTUITION



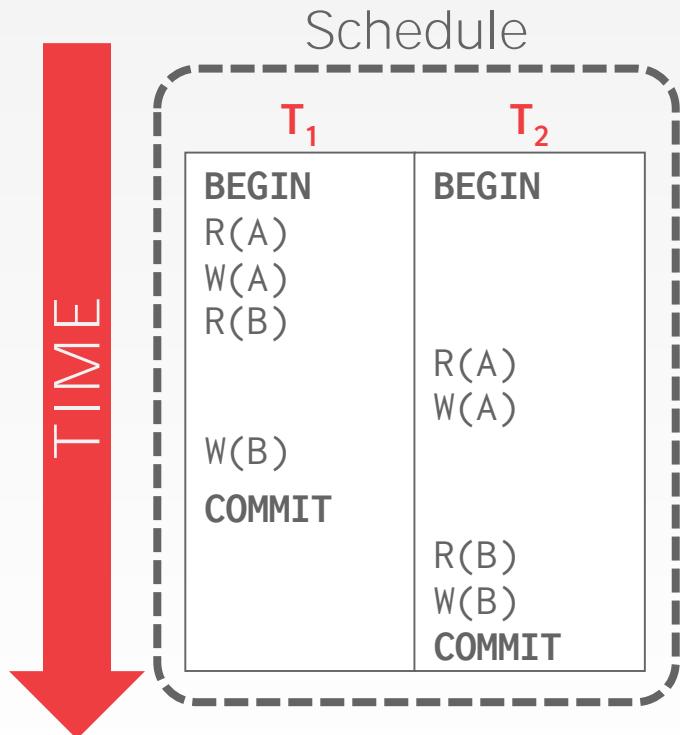
CONFLICT SERIALIZABILITY INTUITION



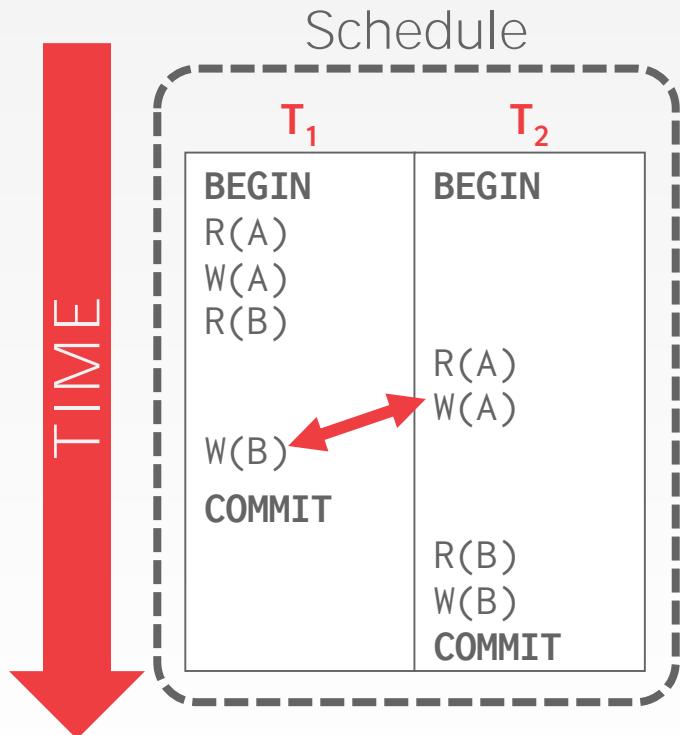
CONFLICT SERIALIZABILITY INTUITION



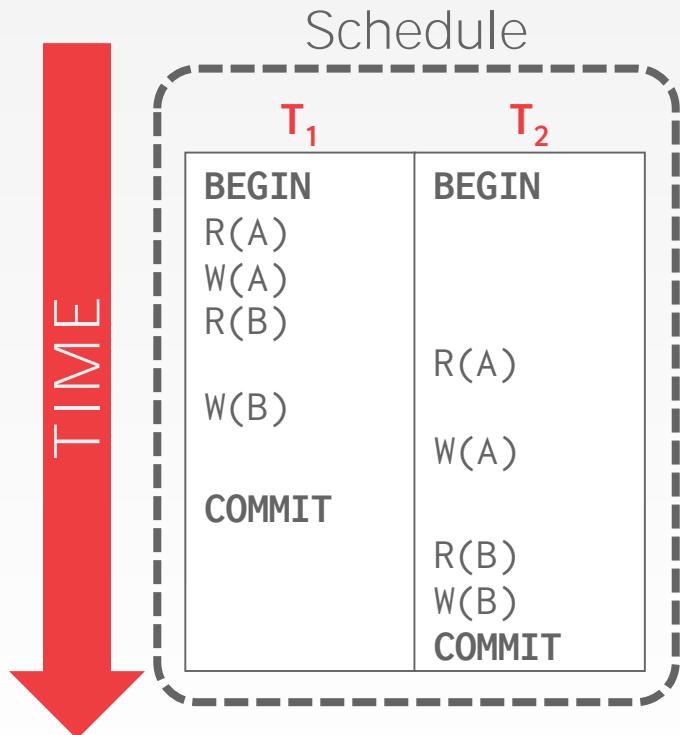
CONFLICT SERIALIZABILITY INTUITION



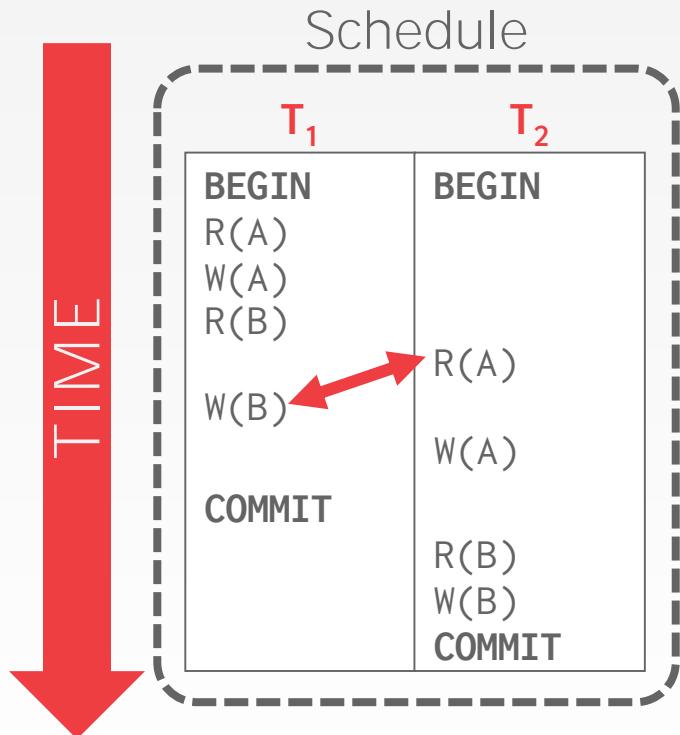
CONFLICT SERIALIZABILITY INTUITION



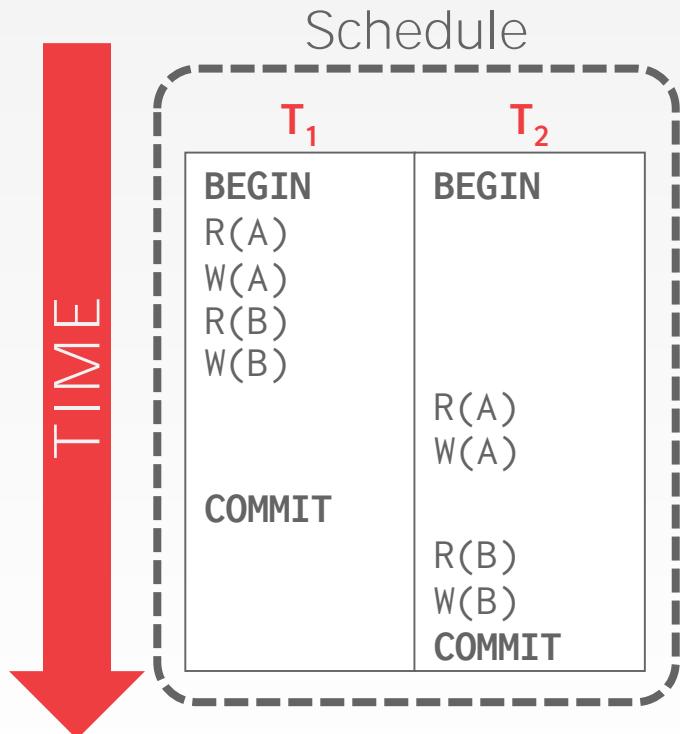
CONFLICT SERIALIZABILITY INTUITION



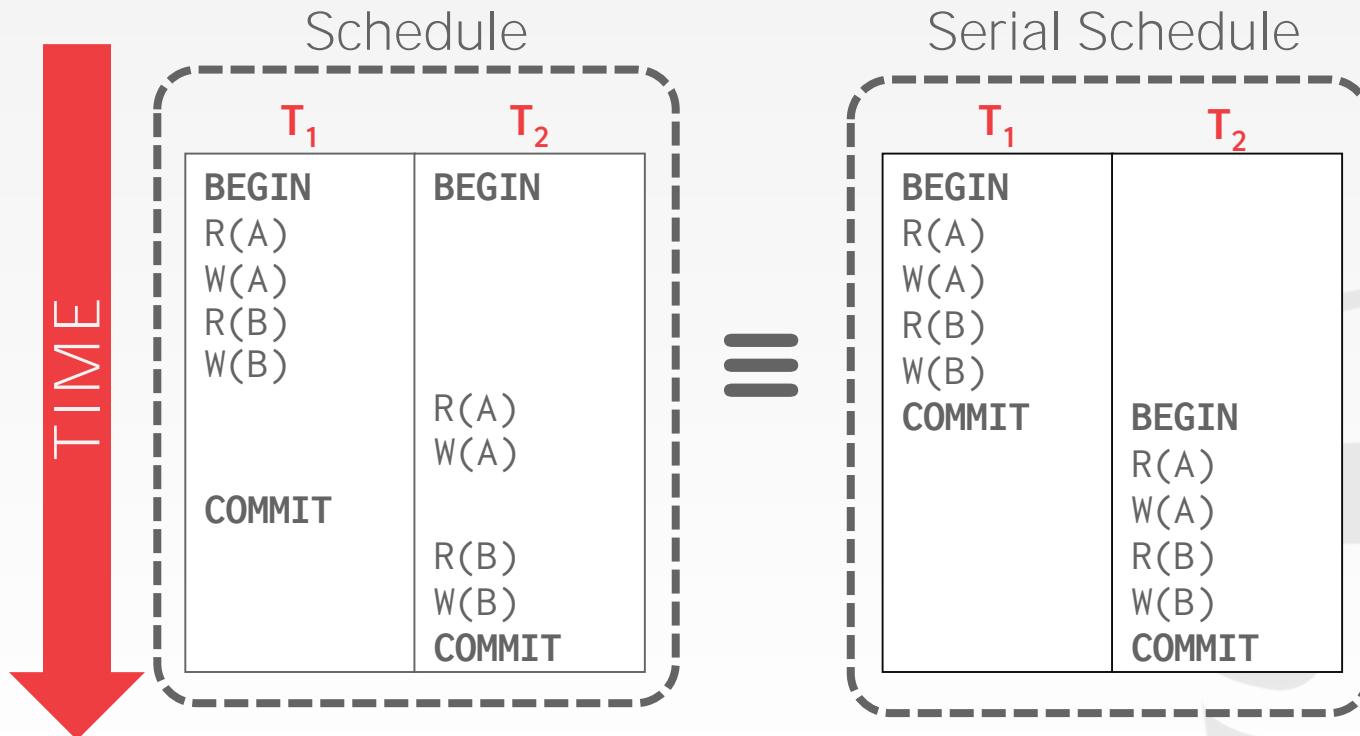
CONFLICT SERIALIZABILITY INTUITION



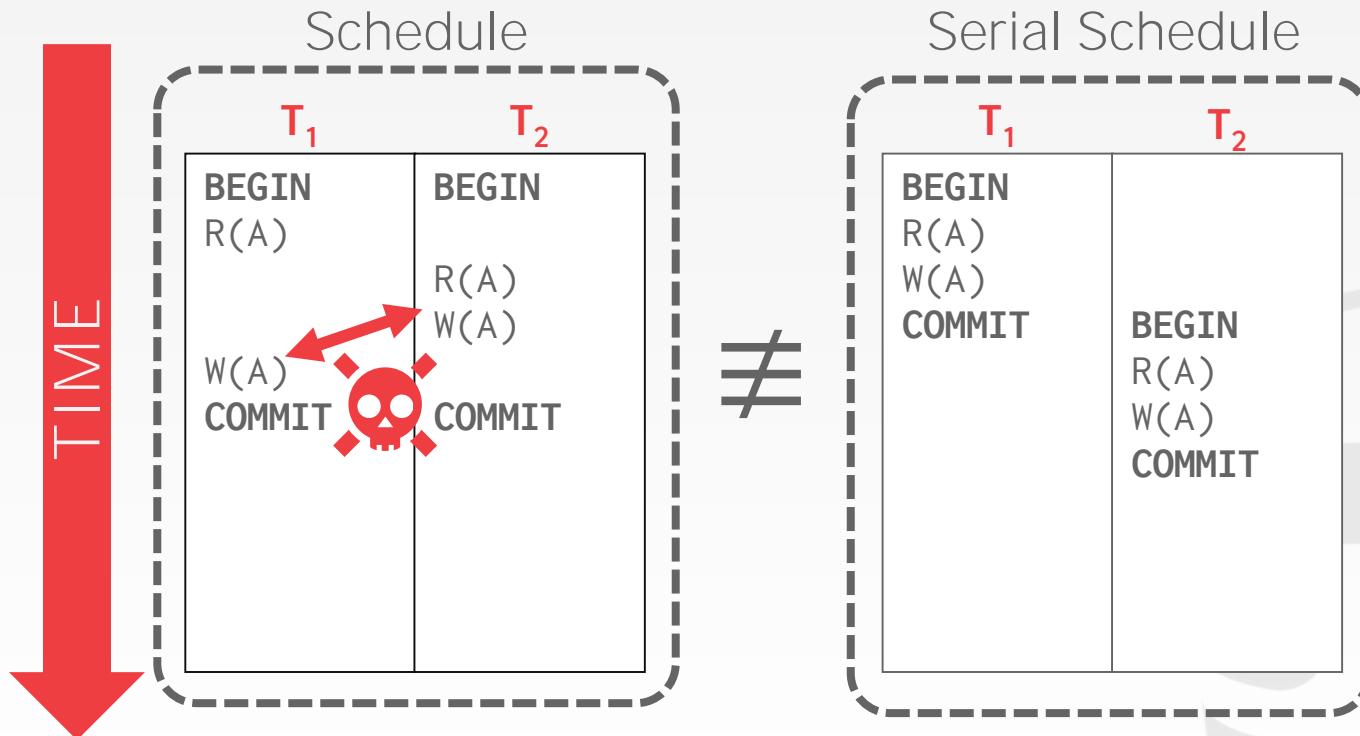
CONFLICT SERIALIZABILITY INTUITION



CONFLICT SERIALIZABILITY INTUITION



CONFLICT SERIALIZABILITY INTUITION



SERIALIZABILITY

Swapping operations is easy when there are only two txns in the schedule. It's cumbersome when there are many txns.

Are there any faster algorithms to figure this out other than transposing operations?



DEPENDENCY GRAPHS

One node per txn.

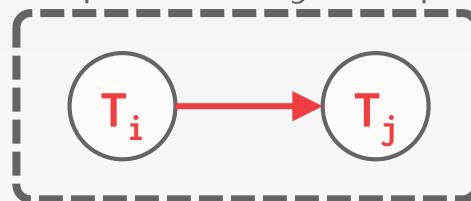
Edge from T_i to T_j if:

- An operation O_i of T_i conflicts with an operation O_j of T_j and
- O_i appears earlier in the schedule than O_j .

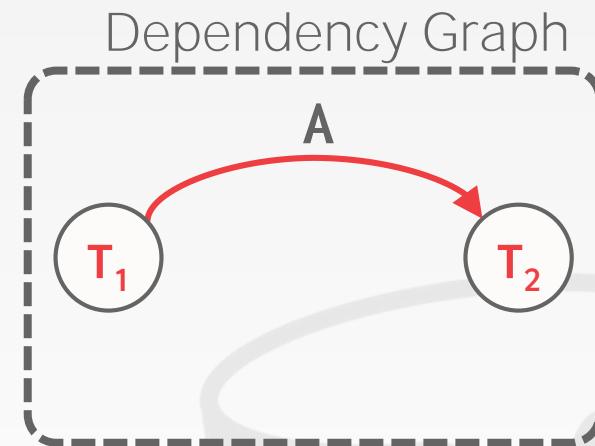
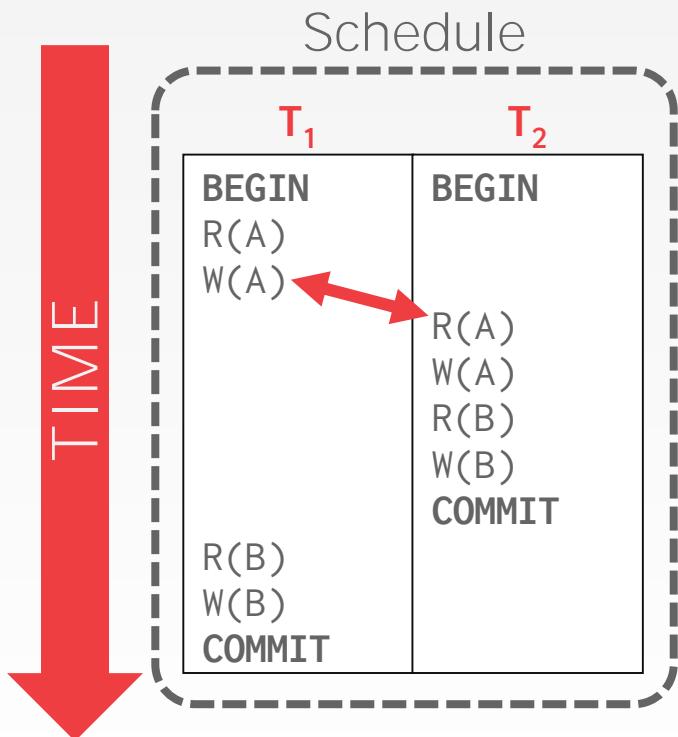
Also known as a precedence graph.

A schedule is conflict serializable iff its dependency graph is acyclic.

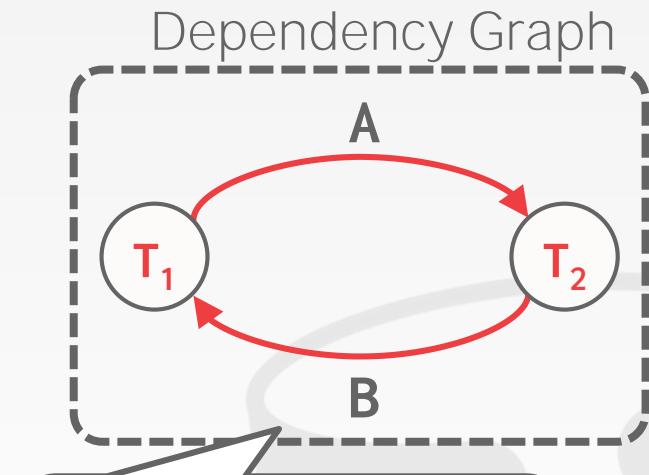
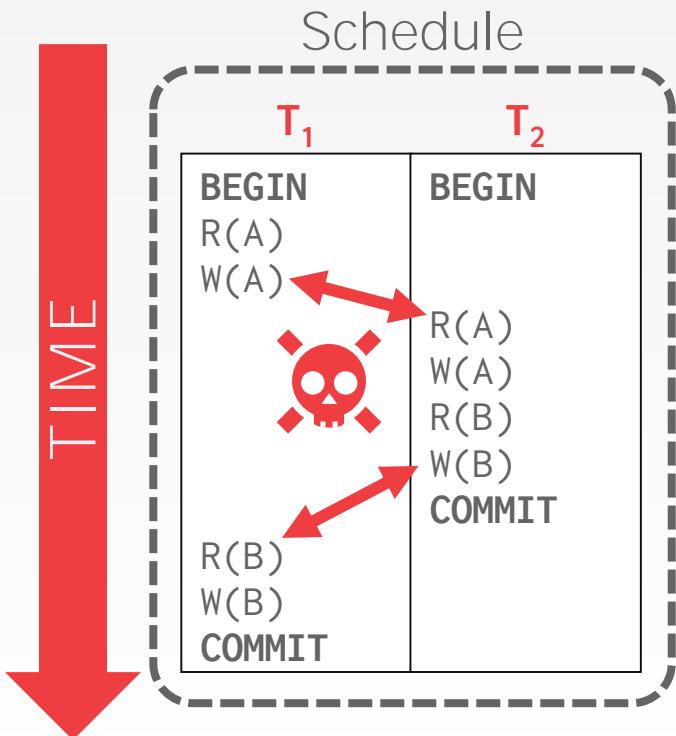
Dependency Graph



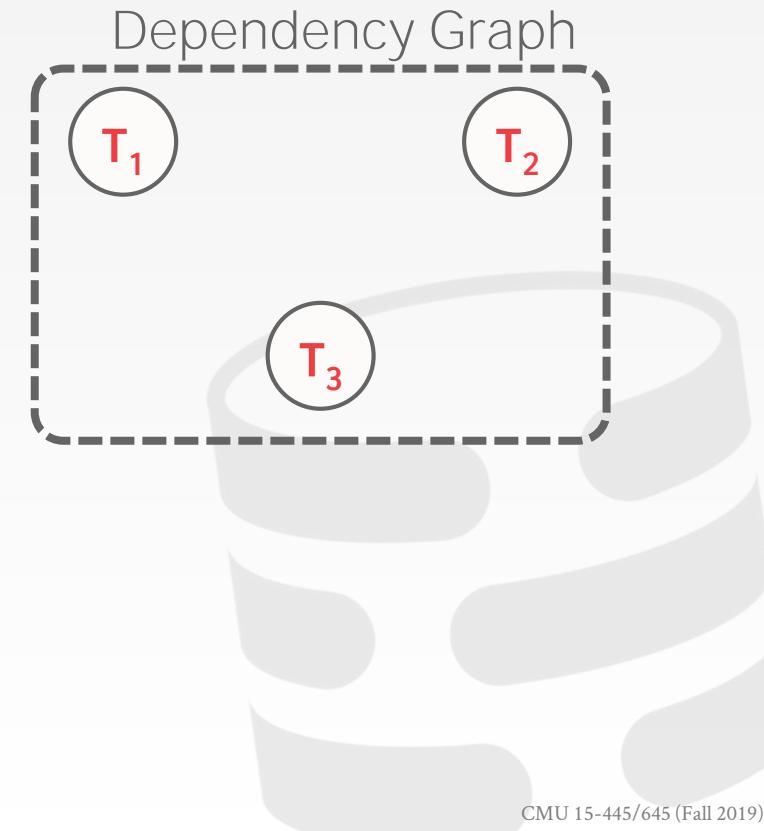
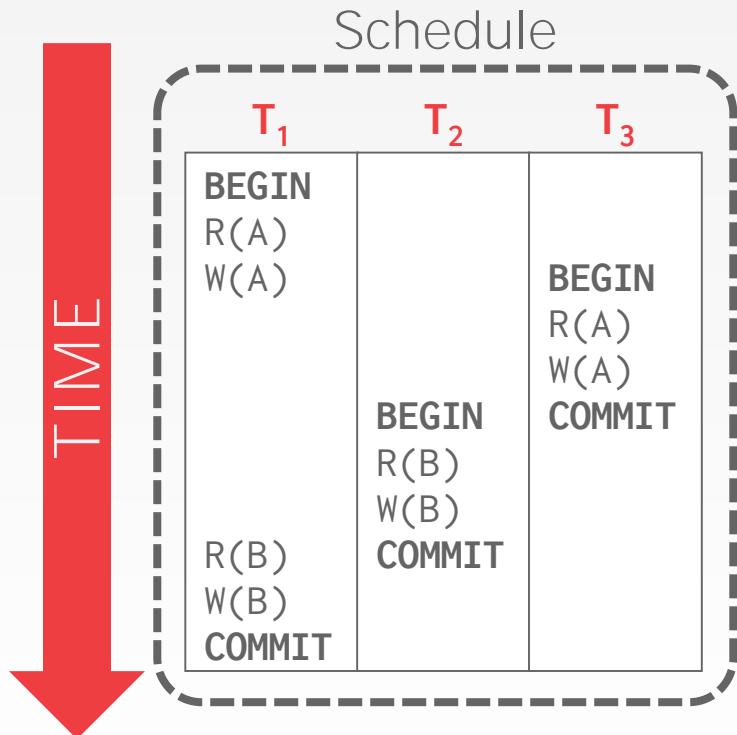
EXAMPLE #1



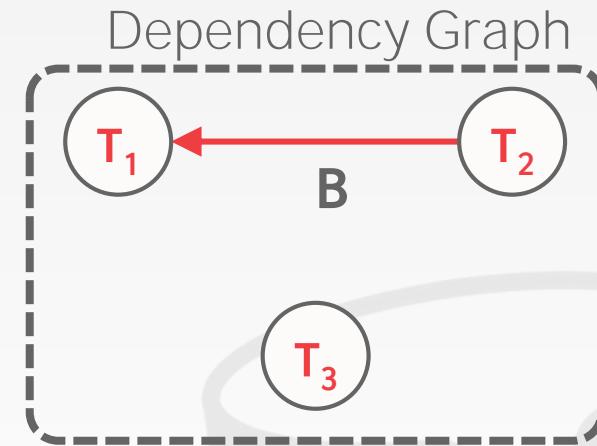
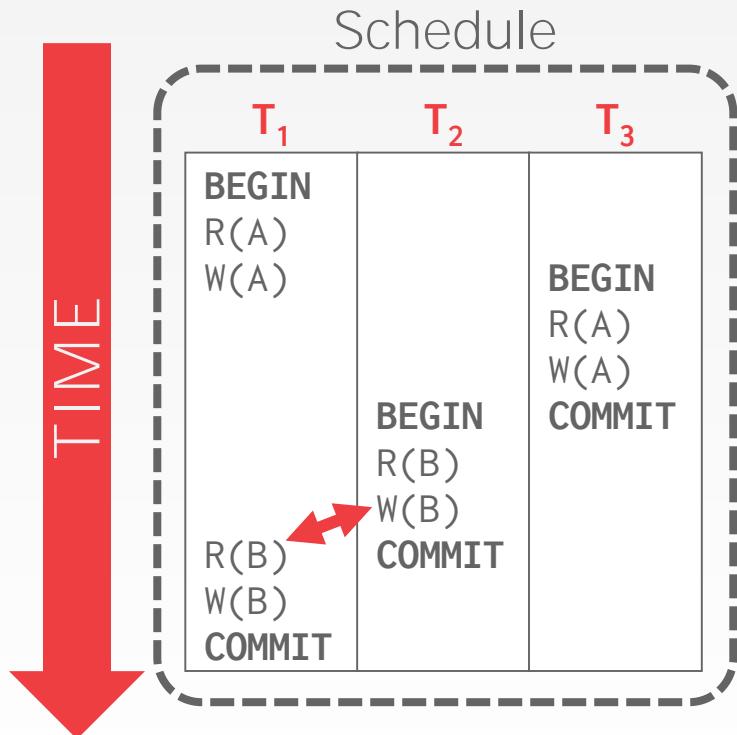
EXAMPLE #1



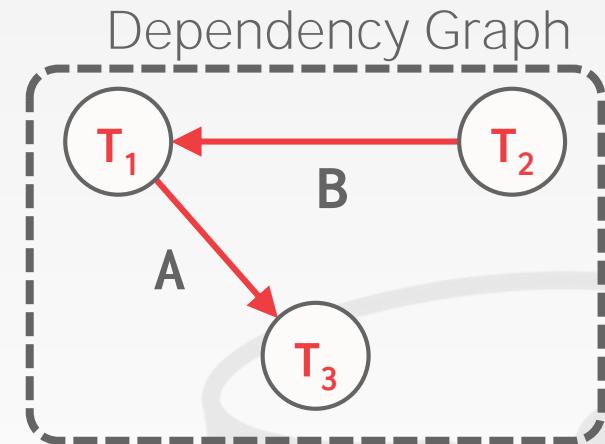
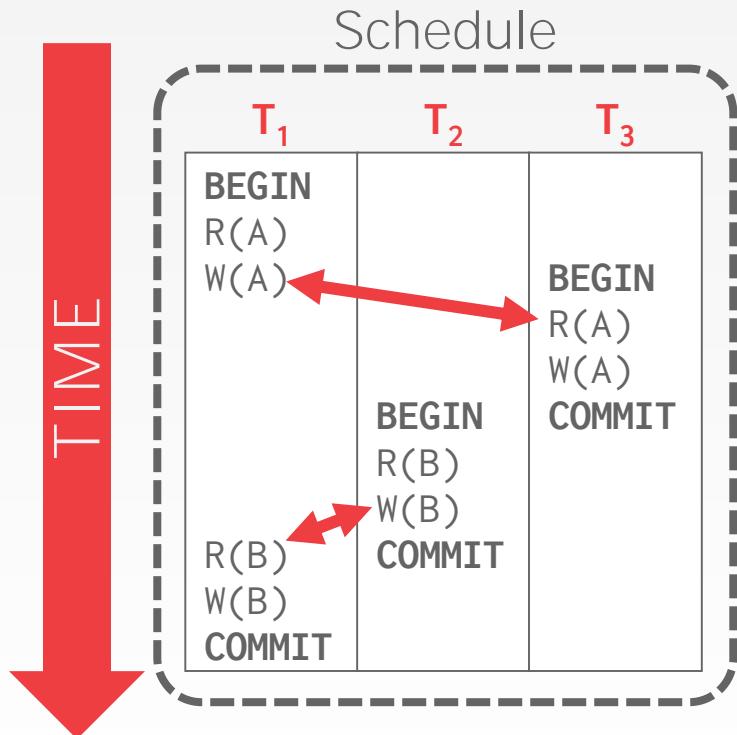
EXAMPLE #2 – THREESOME



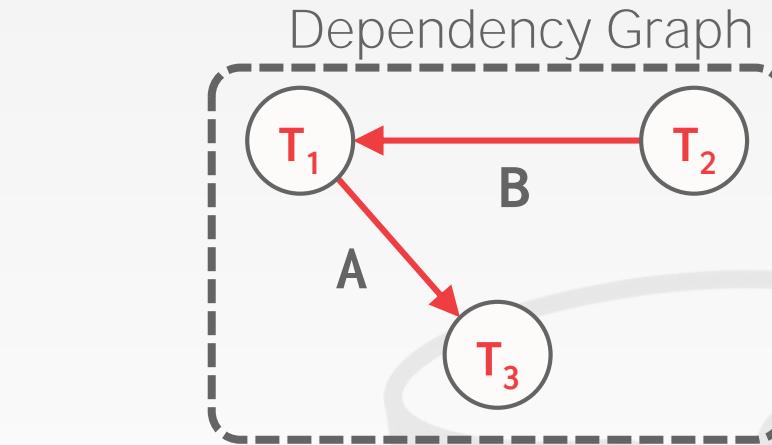
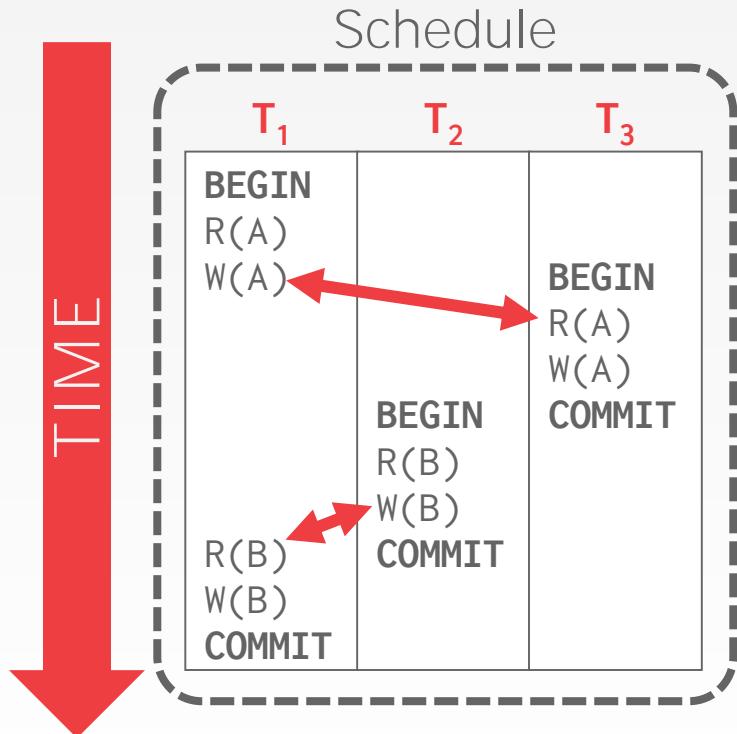
EXAMPLE #2 – THREESOME



EXAMPLE #2 – THREESOME

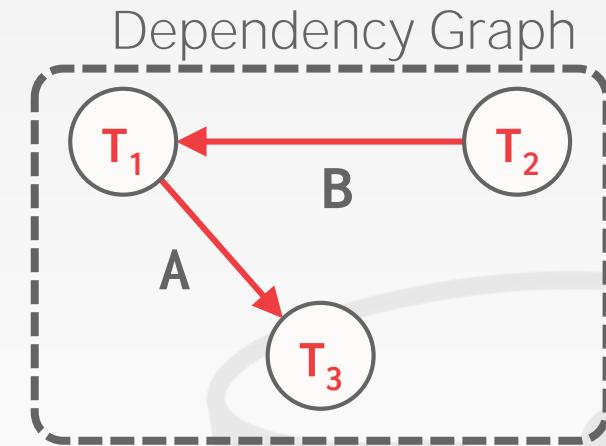
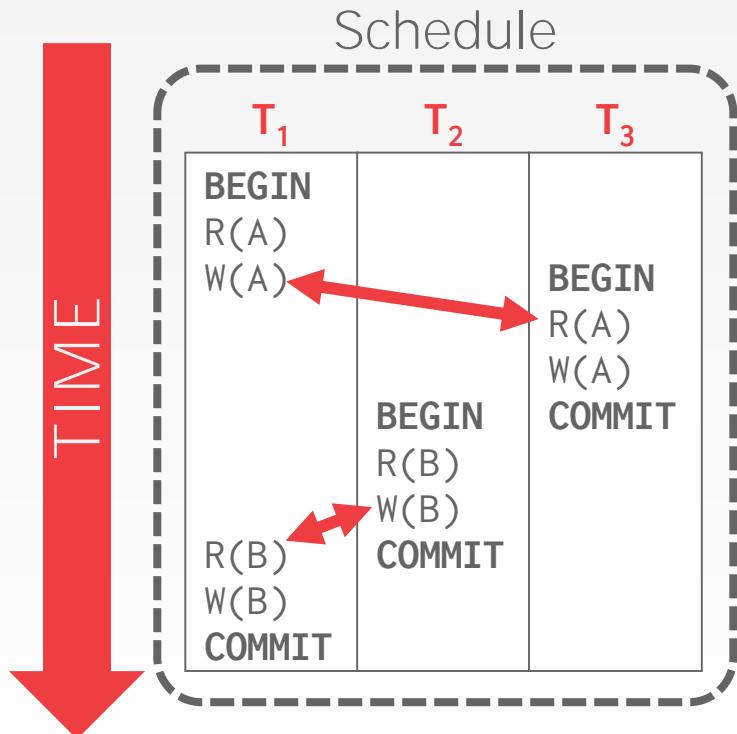


EXAMPLE #2 – THREESOME



Is this equivalent to a serial execution?

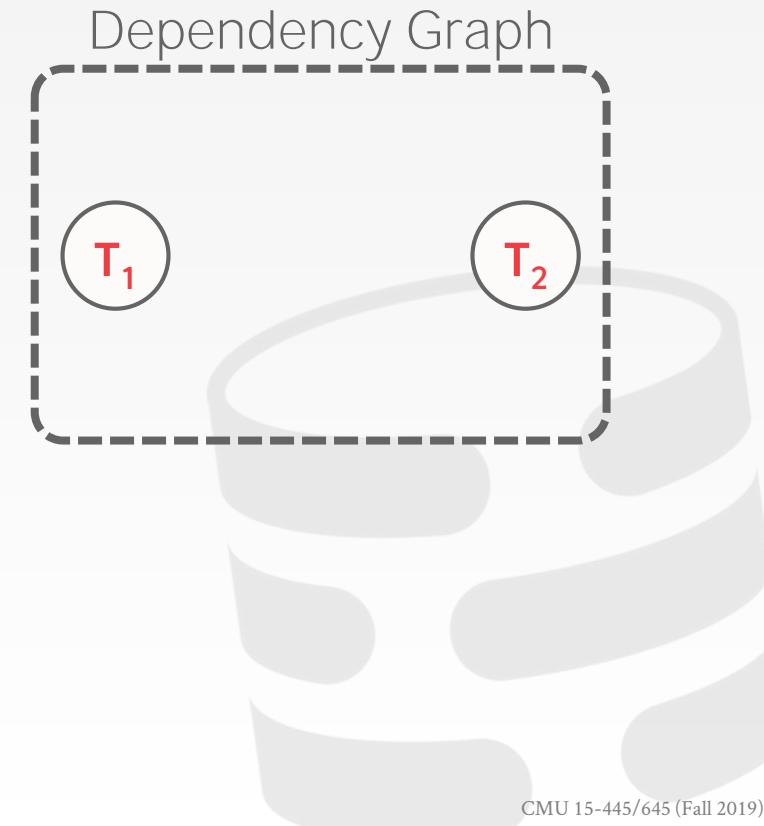
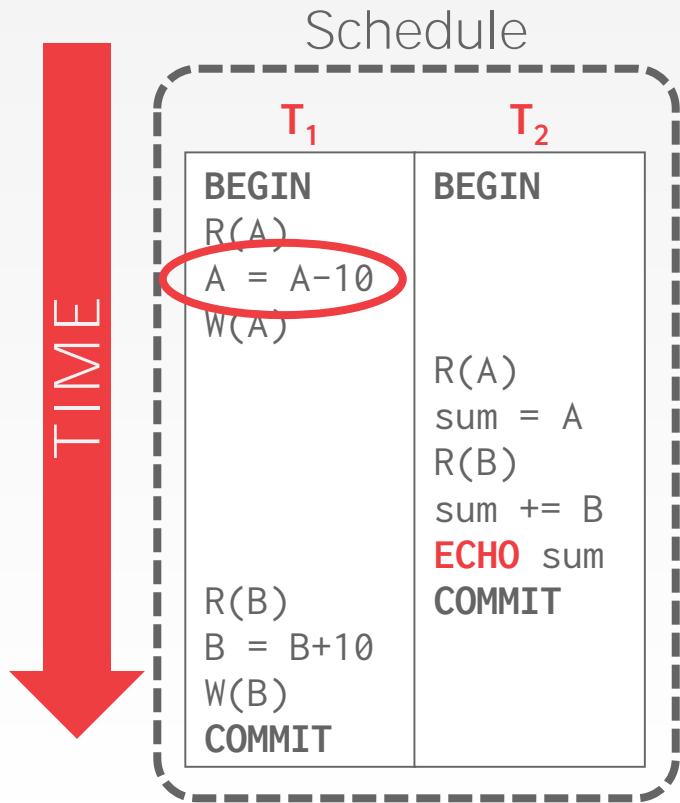
EXAMPLE #2 – THREESOME



Is this equivalent to a serial execution?

Yes (T_2, T_1, T_3)
 → Notice that T_3 should go after T_2 , although it starts before it!

EXAMPLE #3 – INCONSISTENT ANALYSIS

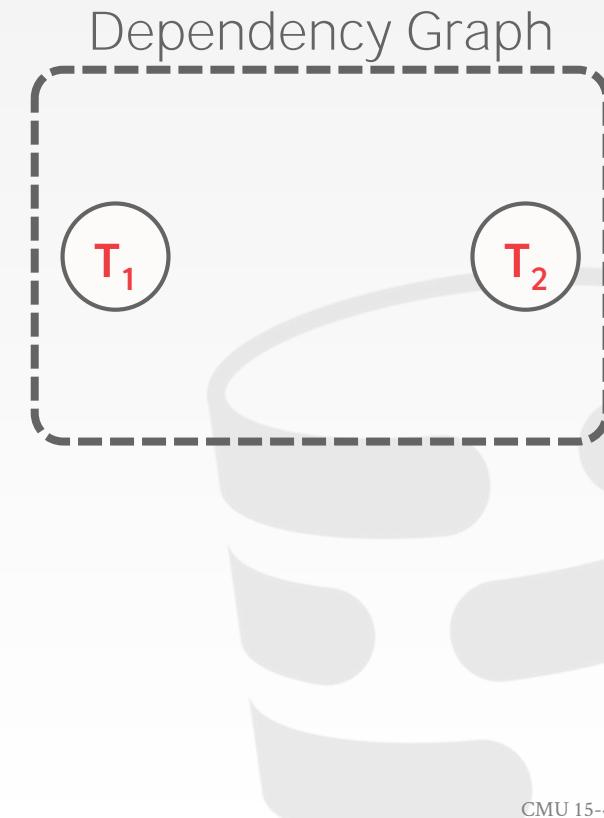


EXAMPLE #3 – INCONSISTENT ANALYSIS

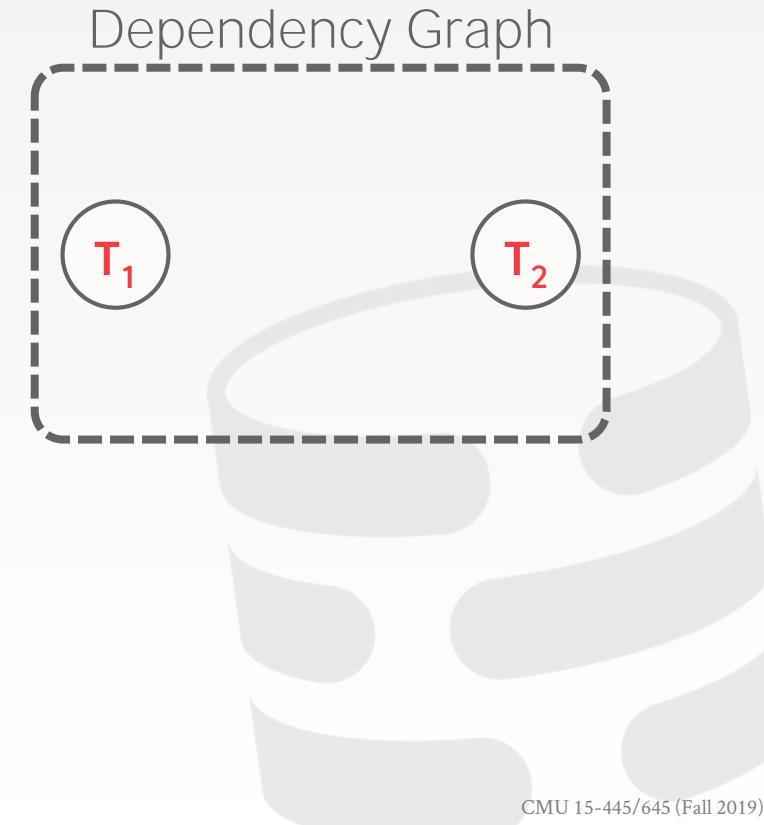
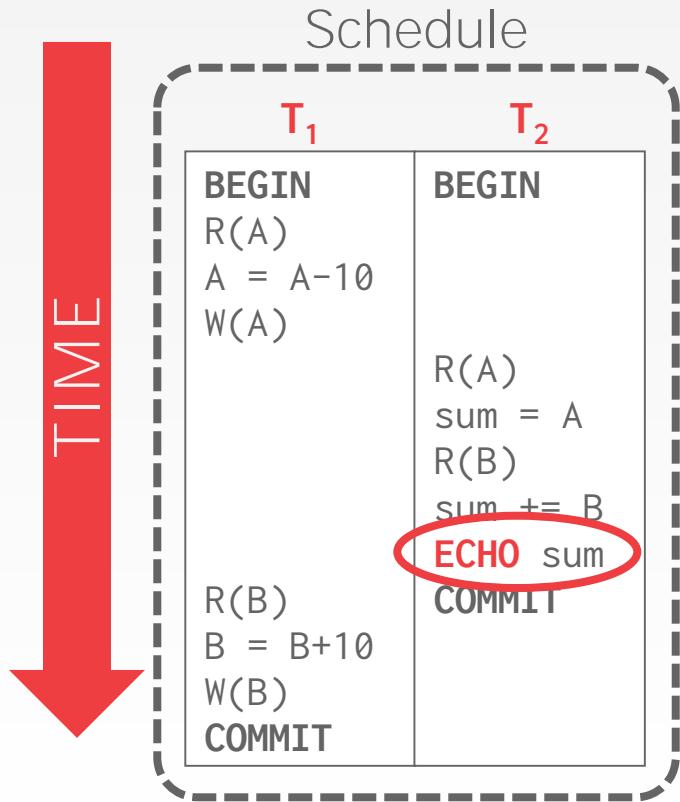


Schedule

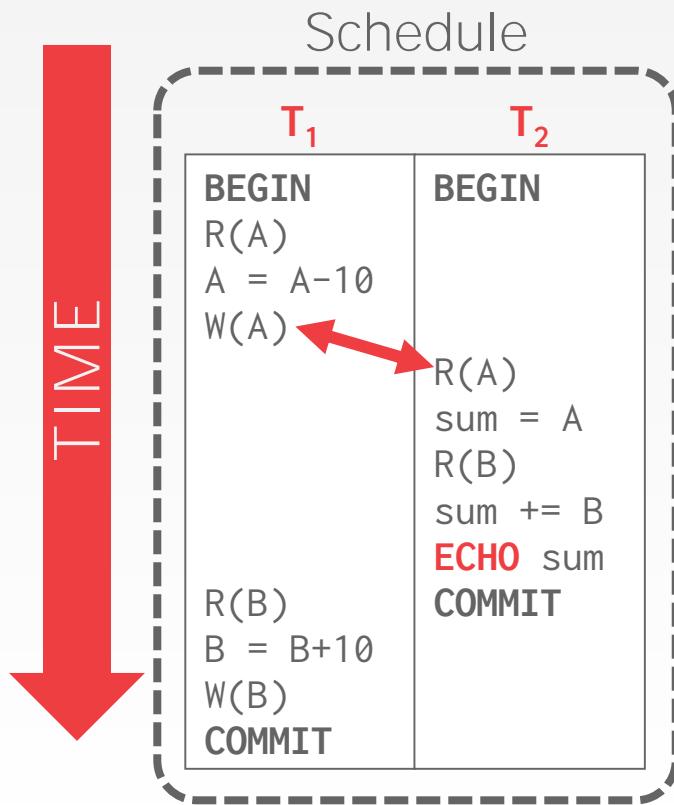
T_1	T_2
BEGIN	BEGIN
$R(A)$	
$A = A - 10$	
$W(A)$	
	$R(A)$
	$sum = A$
	$R(B)$
	$sum += B$
	ECHO sum
	COMMIT
$R(B)$	
$B = B + 10$	
$W(B)$	
COMMIT	



EXAMPLE #3 – INCONSISTENT ANALYSIS

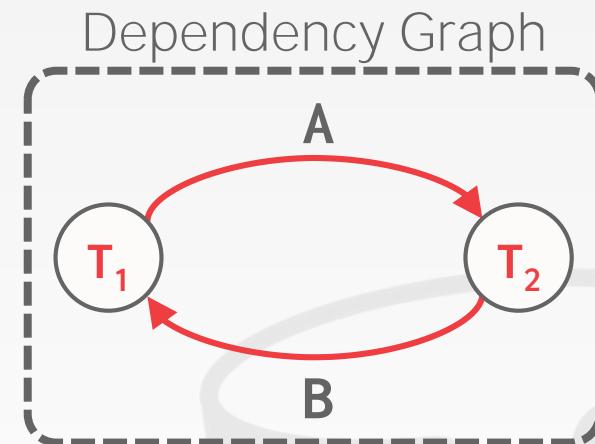
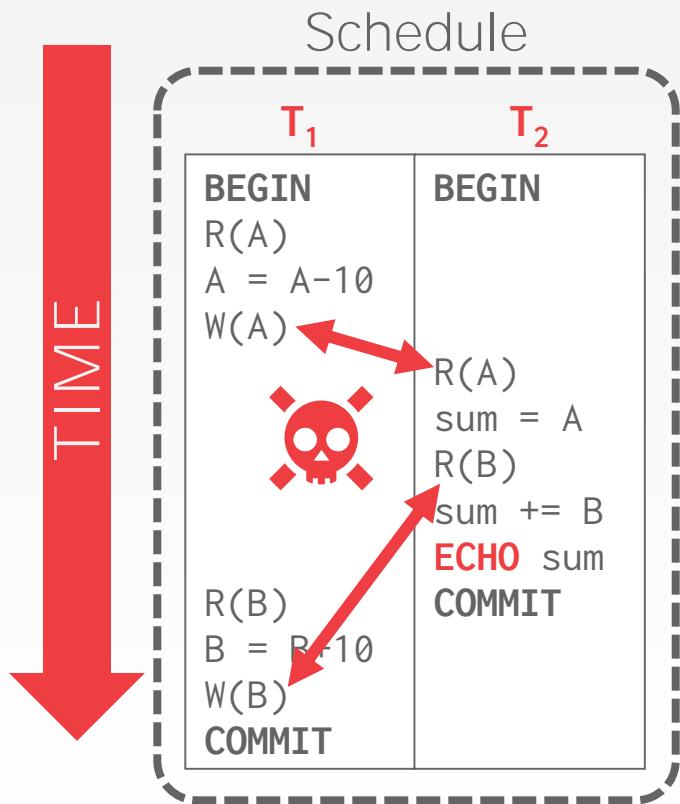


EXAMPLE #3 – INCONSISTENT ANALYSIS

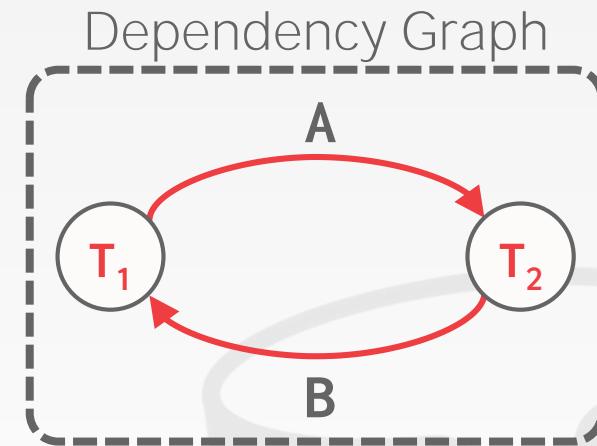
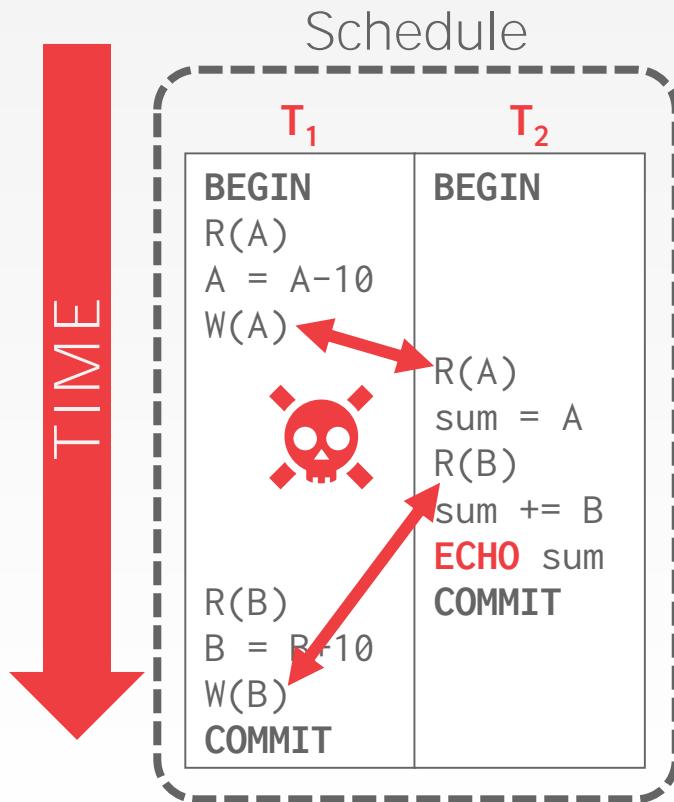


Dependency Graph

EXAMPLE #3 – INCONSISTENT ANALYSIS

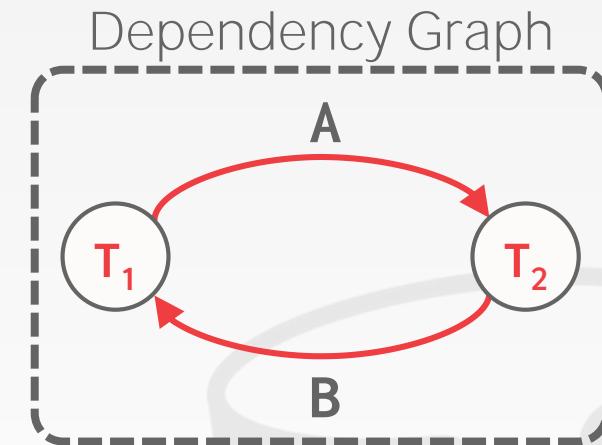
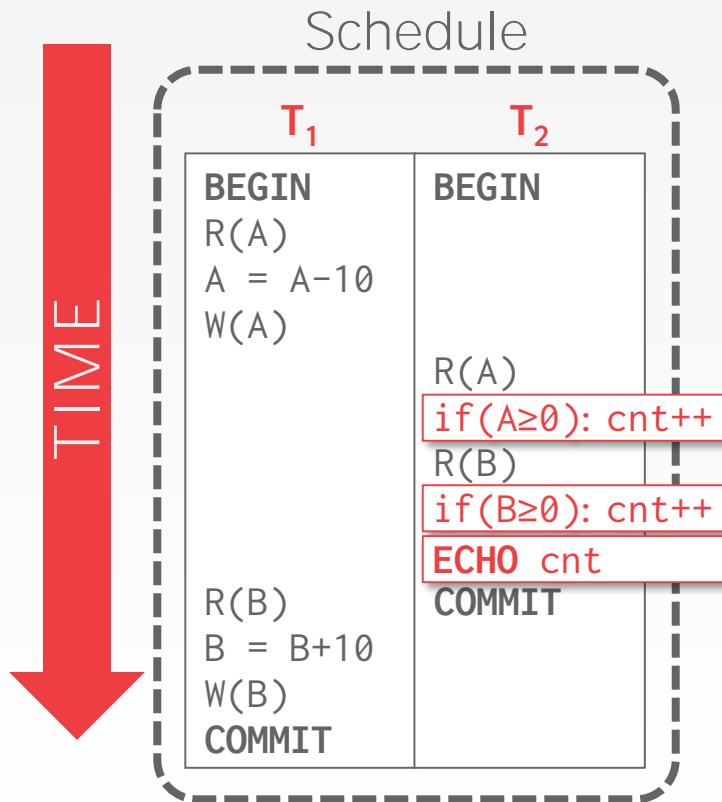


EXAMPLE #3 – INCONSISTENT ANALYSIS



Is it possible to modify only the application logic so that schedule produces a "correct" result but is still not conflict serializable?

EXAMPLE #3 – INCONSISTENT ANALYSIS



Is it possible to modify only the application logic so that schedule produces a "correct" result but is still not conflict serializable?

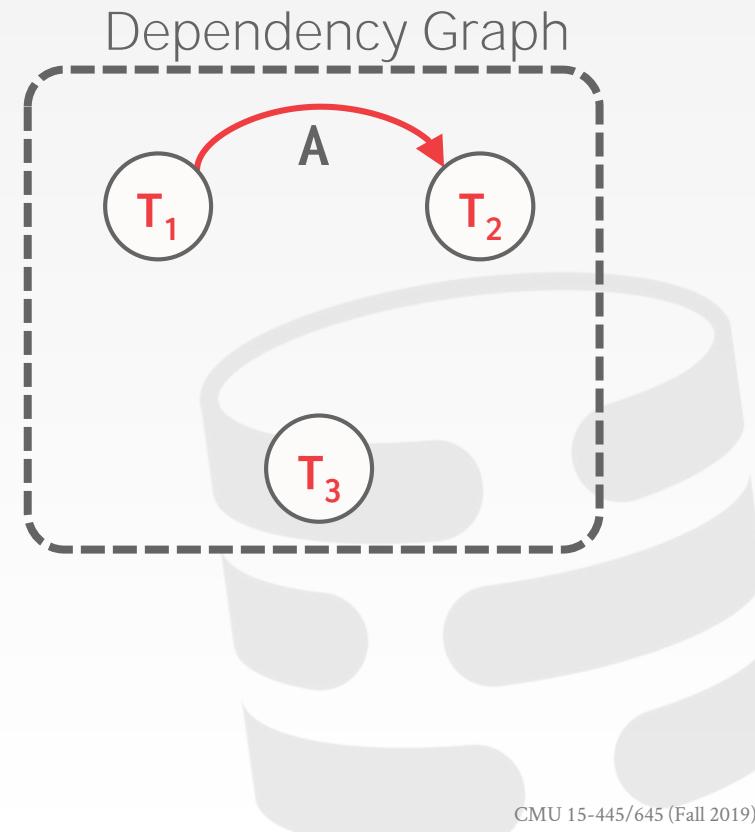
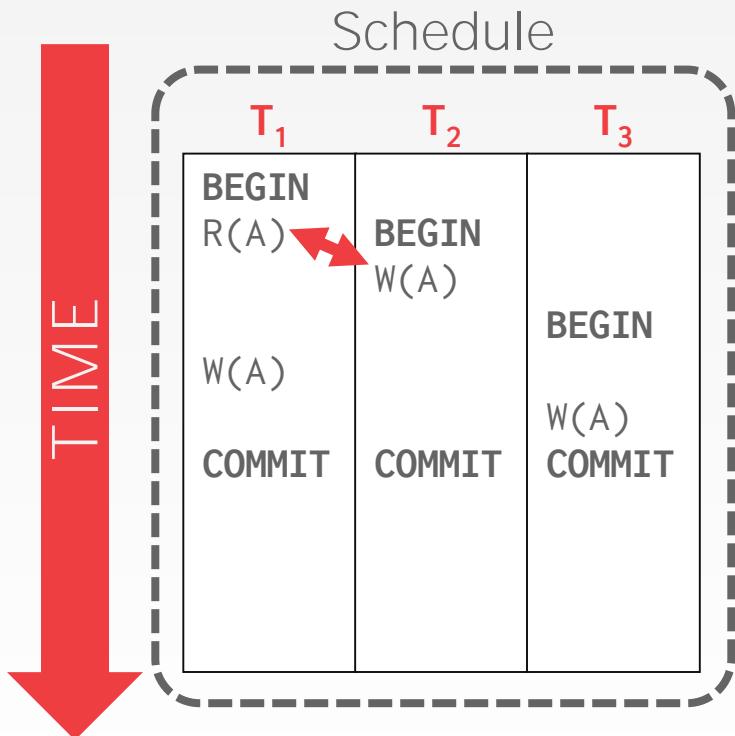
VIEW SERIALIZABILITY

Alternative (weaker) notion of serializability.

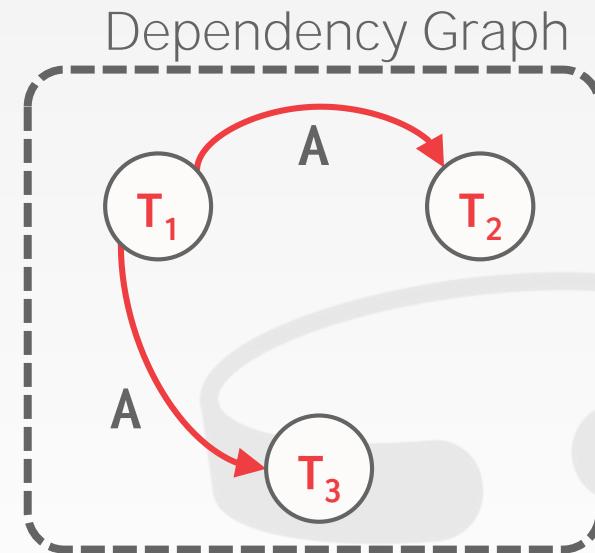
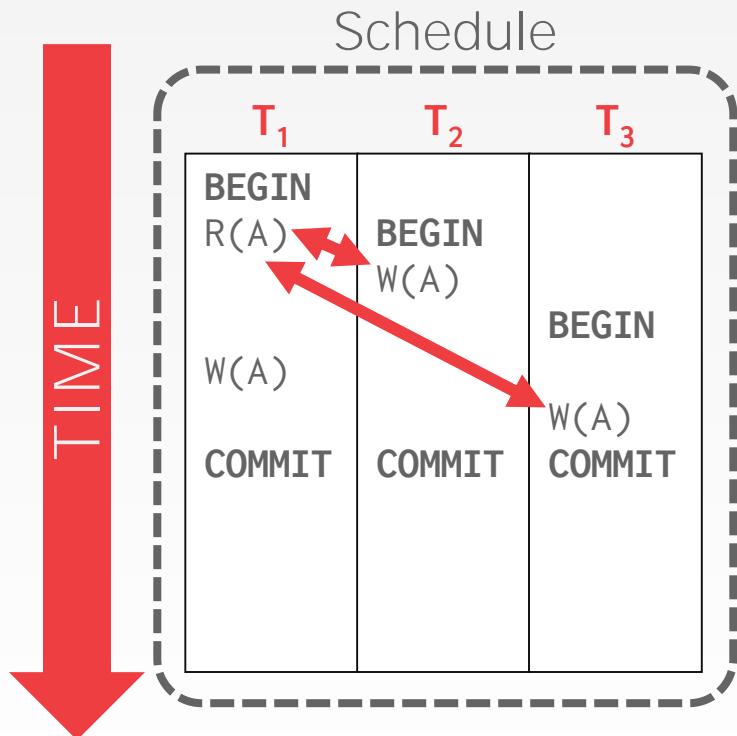
Schedules S_1 and S_2 are view equivalent if:

- If T_1 reads initial value of A in S_1 , then T_1 also reads initial value of A in S_2 .
- If T_1 reads value of A written by T_2 in S_1 , then T_1 also reads value of A written by T_2 in S_2 .
- If T_1 writes final value of A in S_1 , then T_1 also writes final value of A in S_2 .

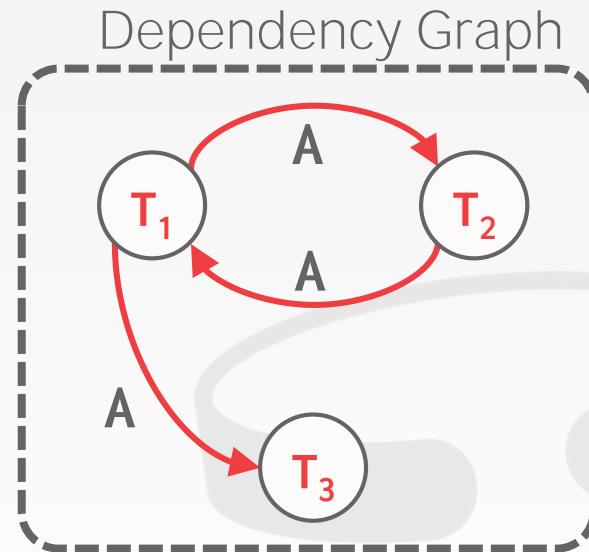
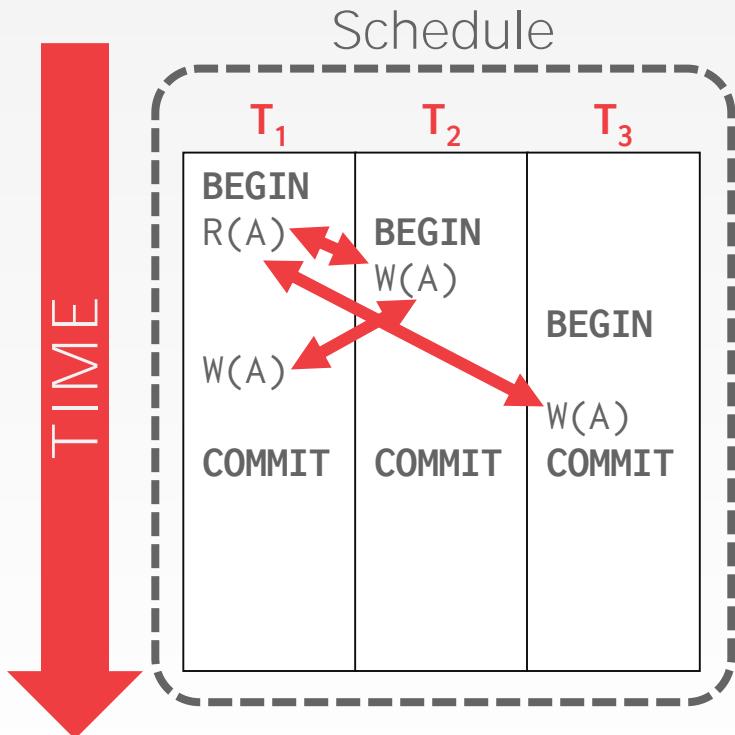
VIEW SERIALIZABILITY



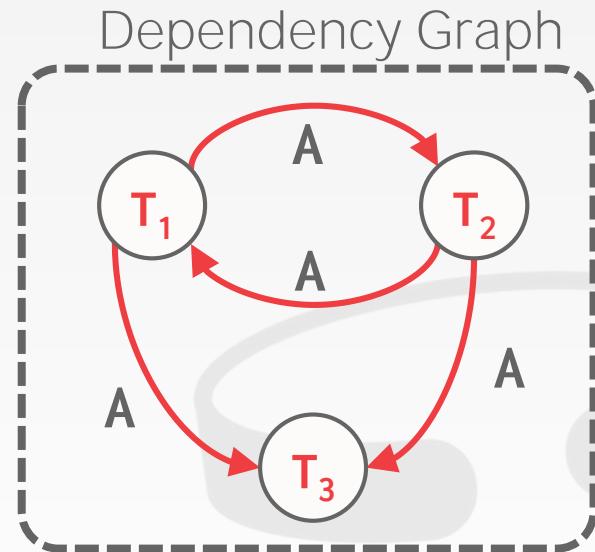
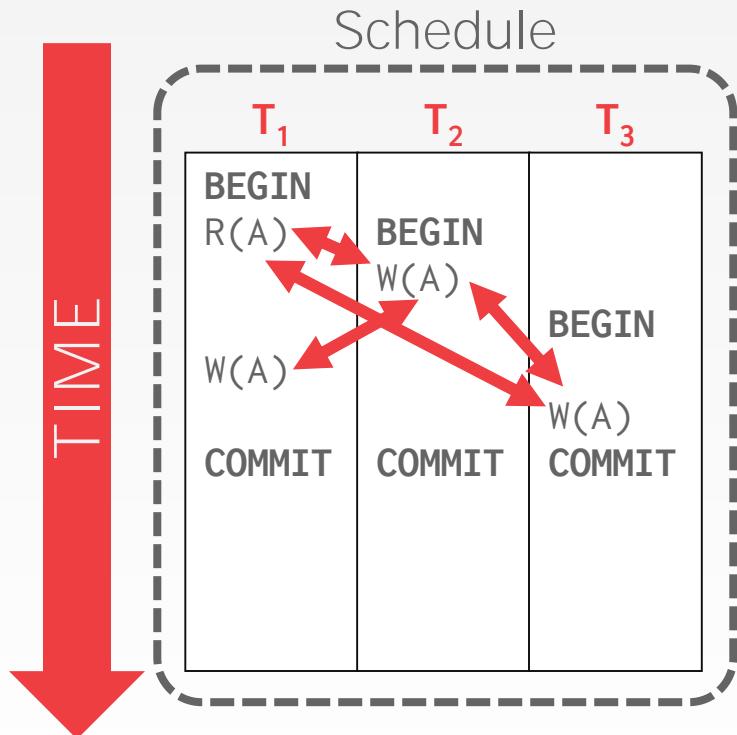
VIEW SERIALIZABILITY



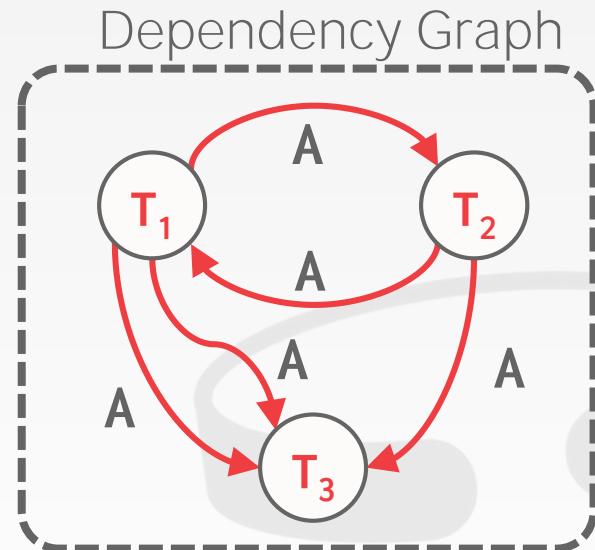
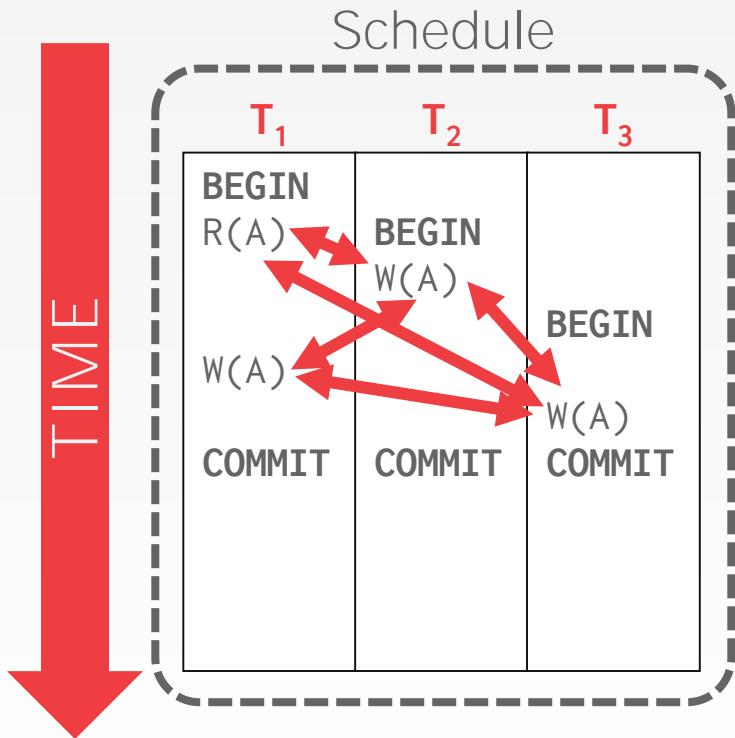
VIEW SERIALIZABILITY



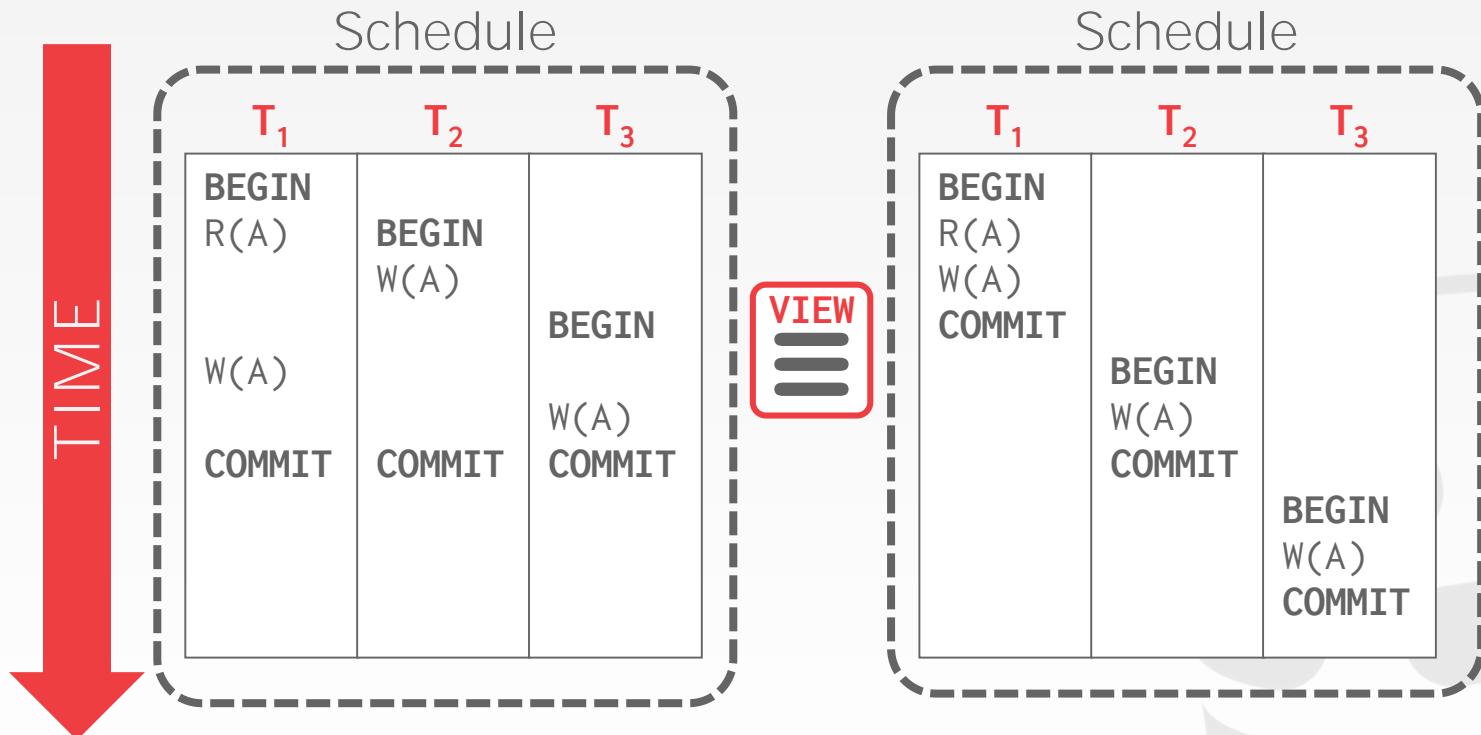
VIEW SERIALIZABILITY



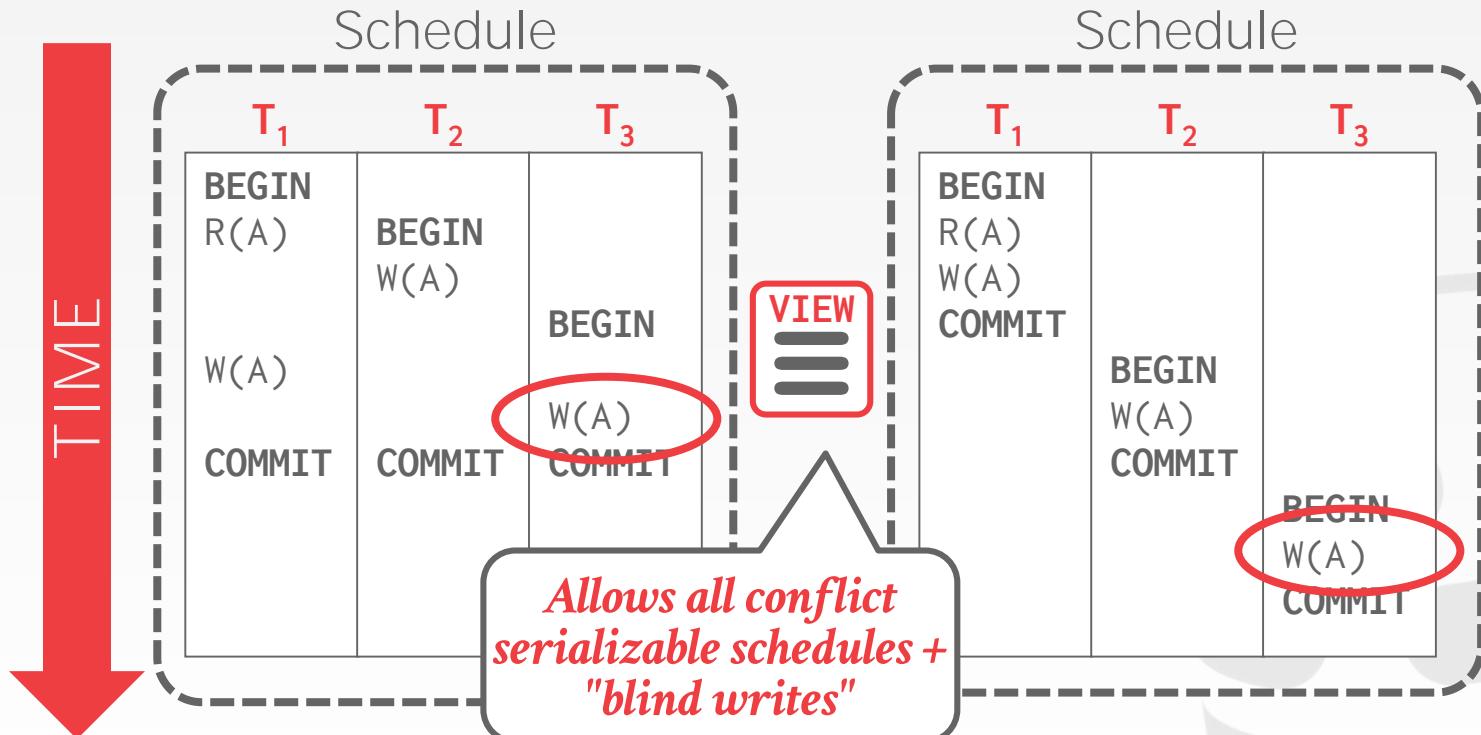
VIEW SERIALIZABILITY



VIEW SERIALIZABILITY



VIEW SERIALIZABILITY



SERIALIZABILITY

View Serializability allows for (slightly) more schedules than **Conflict Serializability** does.
→ But is difficult to enforce efficiently.

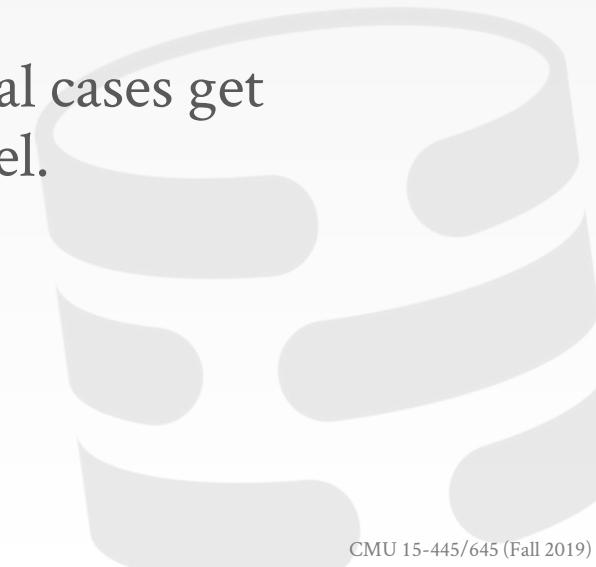
Neither definition allows all schedules that you would consider "serializable".
→ This is because they don't understand the meanings of the operations or the data (recall example #3)



SERIALIZABILITY

In practice, **Conflict Serializability** is what systems support because it can be enforced efficiently.

To allow more concurrency, some special cases get handled separately at the application level.



UNIVERSE OF SCHEDULES

All Schedules

View Serializable

Conflict Serializable

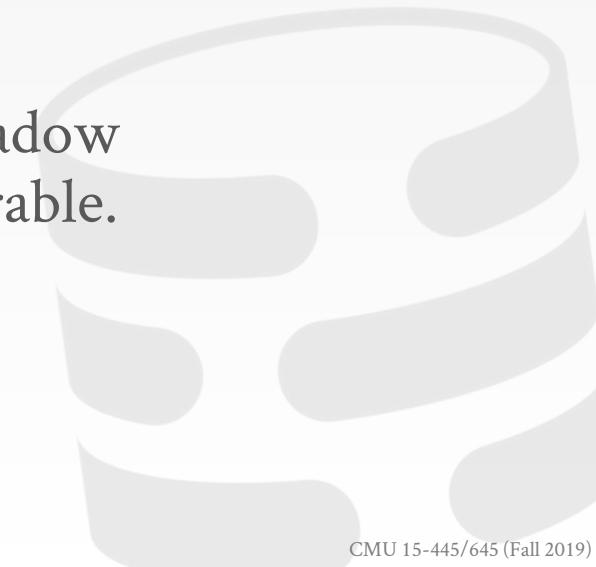
Serial

TRANSACTION DURABILITY

All of the changes of committed transactions should be persistent.

- No torn updates.
- No changes from failed transactions.

The DBMS can use either logging or shadow paging to ensure that all changes are durable.



ACID PROPERTIES

Atomicity: All actions in the txn happen, or none happen.

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one txn is isolated from that of other txns.

Durability: If a txn commits, its effects persist.

CONCLUSION

Concurrency control and recovery are among the most important functions provided by a DBMS.

Concurrency control is automatic

- System automatically inserts lock/unlock requests and schedules actions of different txns.
- Ensures that resulting execution is equivalent to executing the txns one after the other in some order.

CONCLUS

Concurrency control and recovery
most important functions provided by the system

Concurrency control is automatic
→ System automatically inserts locks

ability problems that it brings [9, 10, 19]. 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. Running two-phase commit over Paxos

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [8] because of its semi-relational data model and support for synchronous replication; its relatively poor write throughput. As a result, Spanner has evolved from a Bigtable-like key-value store into a temporal multi-version system. Data is stored in schematized semi-relational tables, which are versioned, and each version is automatically stamped with its commit time; old versions of objects are subject to configurable garbage-collection policies. Spanner supports general-purpose transactions, and provides a query language.

Spanner is a distributed database, Spanner provides interesting features. First, the replication configuration of data can be dynamically controlled at a global level. Applications can specify constraints on which datacenters contain which data, as well as controls from its users (to control read latency), as well as controls from each other (to control write latency). Second, Spanner maintains a consistent number of replicas (to control availability, and read performance). Data is dynamically and transparently moved between datacenters by the system to balance resource usage. Finally, Spanner has two features that are unique to it: it is designed to implement in a distributed database; it is designed to be highly available and consistent.

CONCLUSION

Concurrency control and recovery are among the most important functions provided by a DBMS.

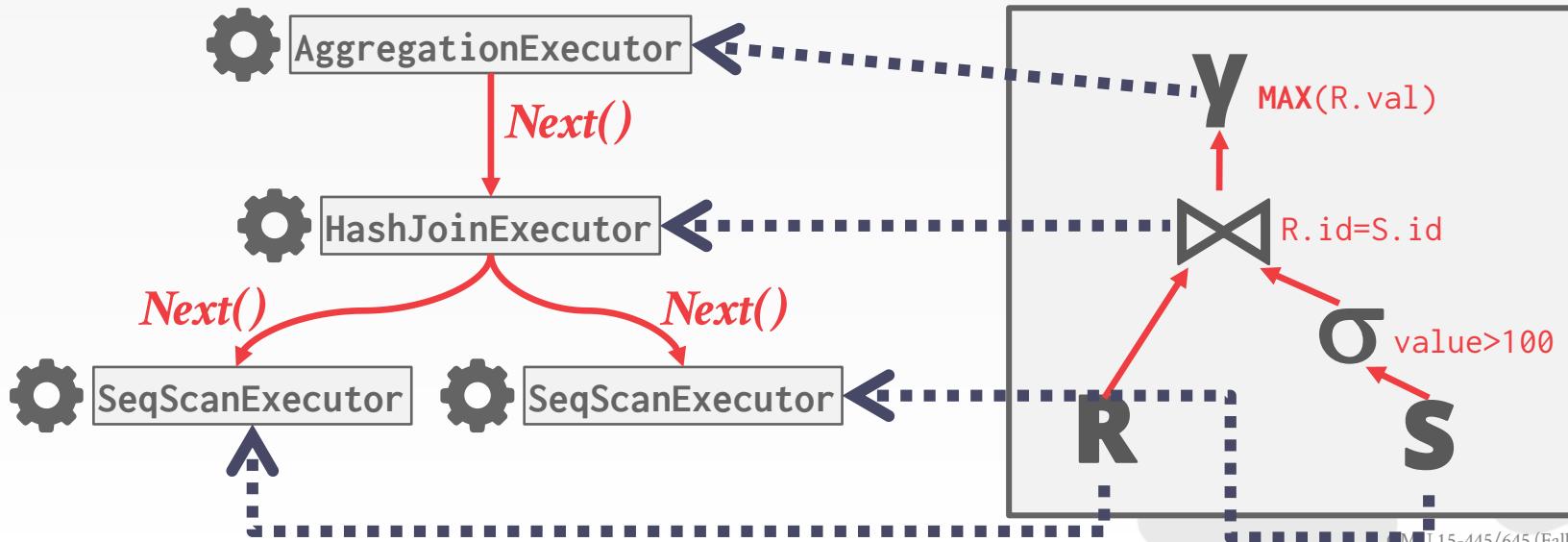
Concurrency control is automatic

- System automatically inserts lock/unlock requests and schedules actions of different txns.
- Ensures that resulting execution is equivalent to executing the txns one after the other in some order.

PROJECT #3

You will build a query execution engine in your DBMS.

```
SELECT MAX(R.val)
  FROM R JOIN S
    ON R.id = S.id
   WHERE S.value > 100
```



PROJECT #3 – TASKS

Install Tables in Catalog

Plan Node Executors

- Insert
- Sequential Scan
- Hash Join
- Hash Aggregation

<https://15445.courses.cs.cmu.edu/fall2019/project2/>

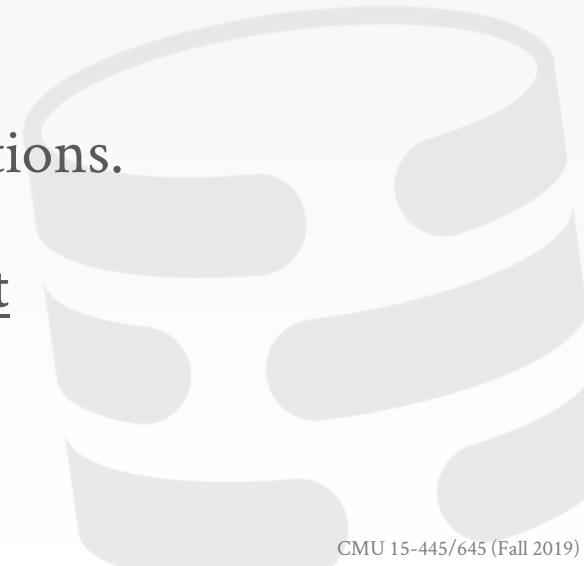
DEVELOPMENT HINTS

You do not need a working Linear Probe Hash Table to complete Tasks #1 and #2.

Implement the insert executor first.

You do not need to worry about transactions.

Gradescope is meant for grading, not debugging. Write your own local tests.



THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Rebase on top of the latest BusTub master branch.

Post your questions on Piazza or come to TA office hours.

PLAGIARISM WARNING

Your project implementation must be your own work.

- You may not copy source code from other groups or the web.
- Do not publish your implementation on Github.

Plagiarism will not be tolerated.

See [CMU's Policy on Academic Integrity](#) for additional information.



NEXT CLASS

Two-Phase Locking
Isolation Levels



17

Two-Phase Locking



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

LAST CLASS

Conflict Serializable

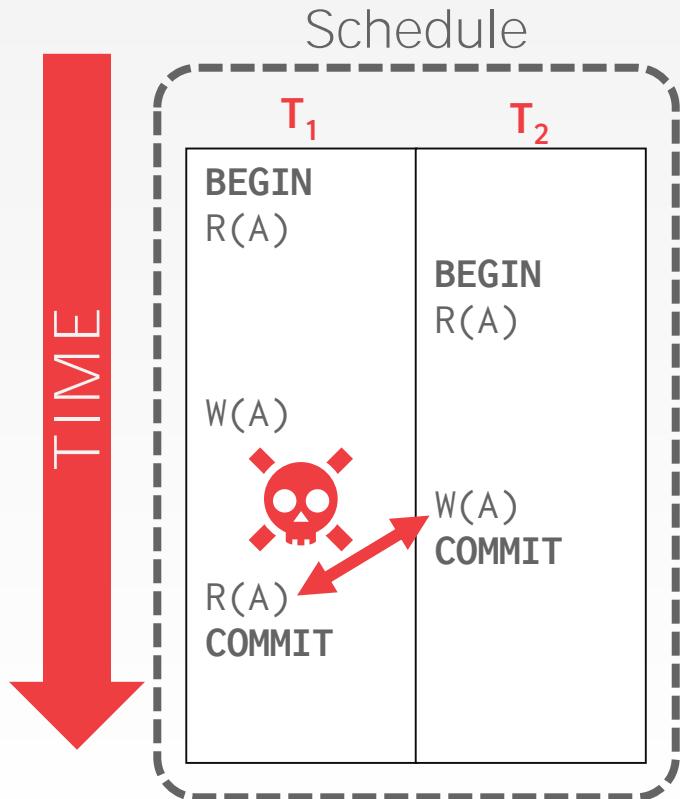
- Verify using either the "swapping" method or dependency graphs.
- Any DBMS that says that they support "serializable" isolation does this.

View Serializable

- No efficient way to verify.
- Andy doesn't know of any DBMS that supports this.



EXAMPLE



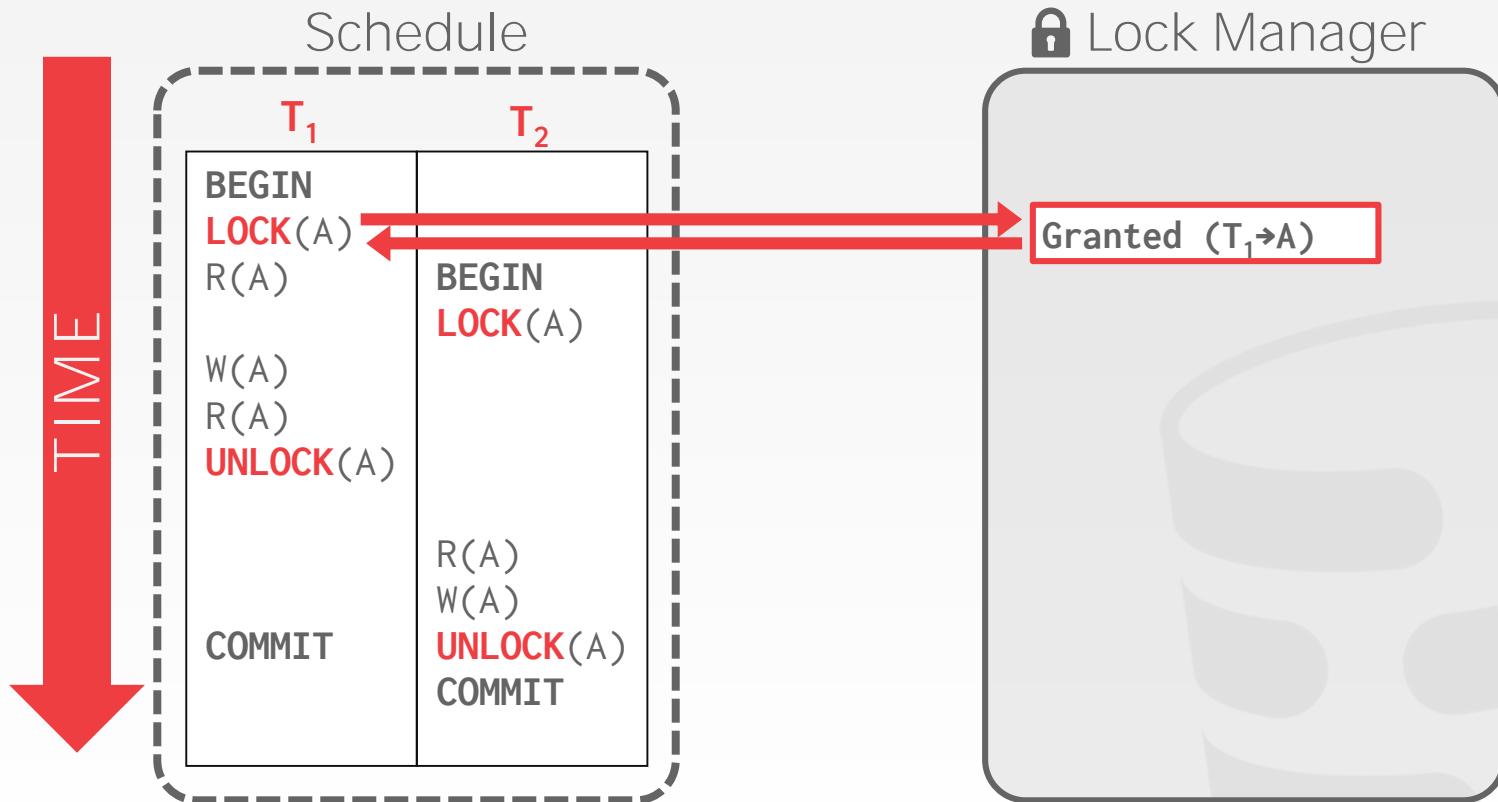
OBSERVATION

We need a way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time.

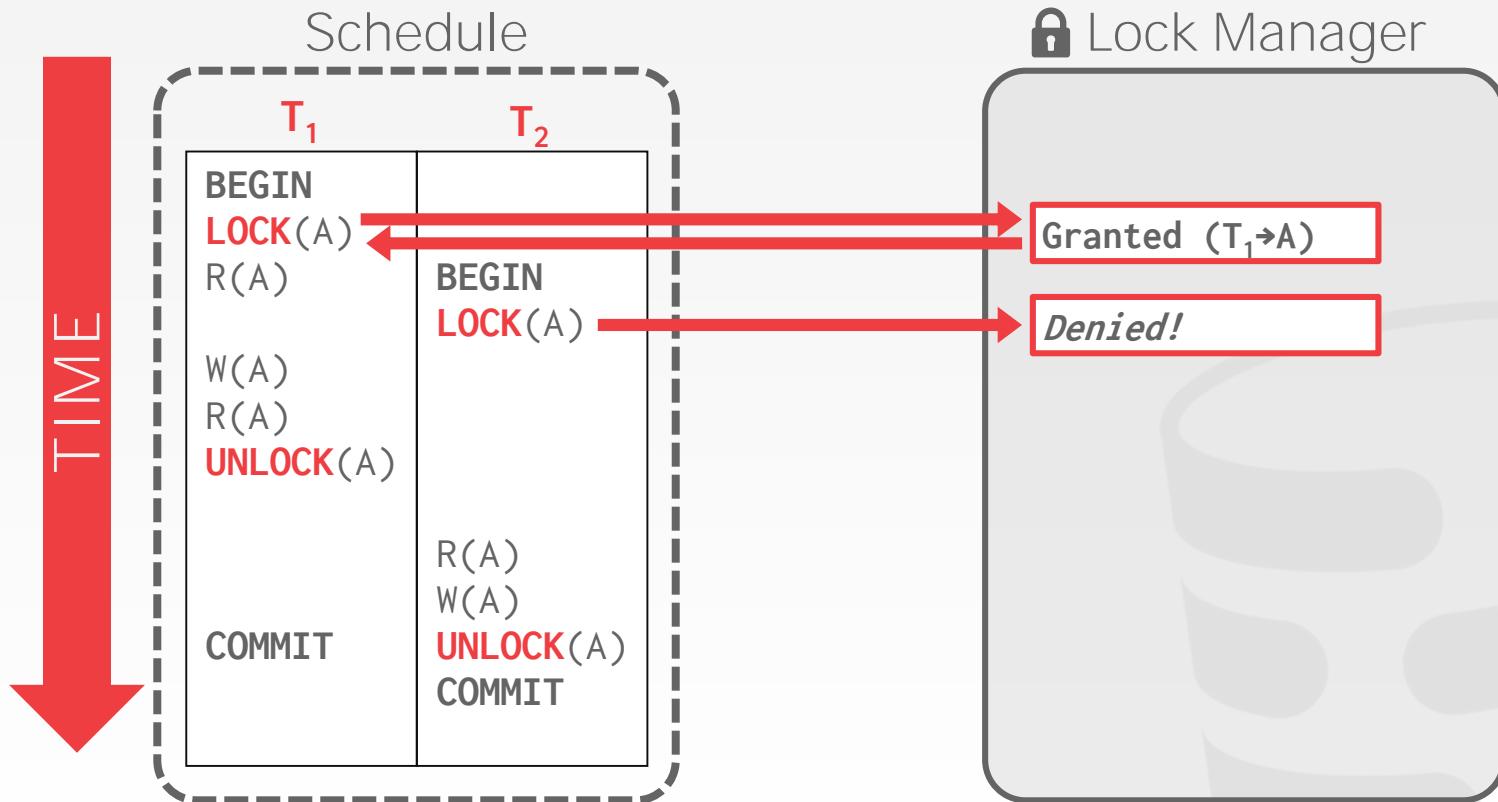
Solution: Use **locks** to protect database objects.



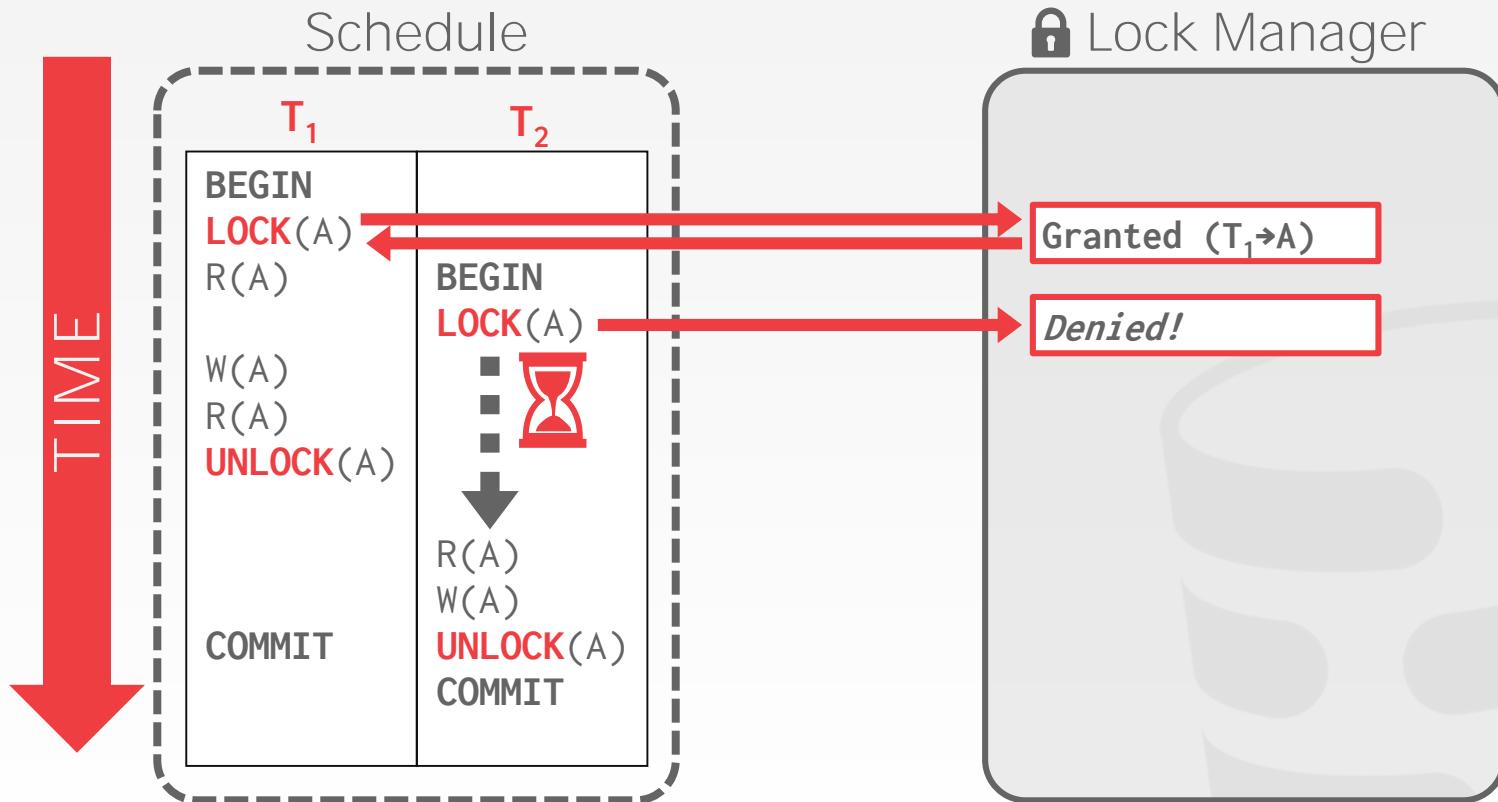
EXECUTING WITH LOCKS



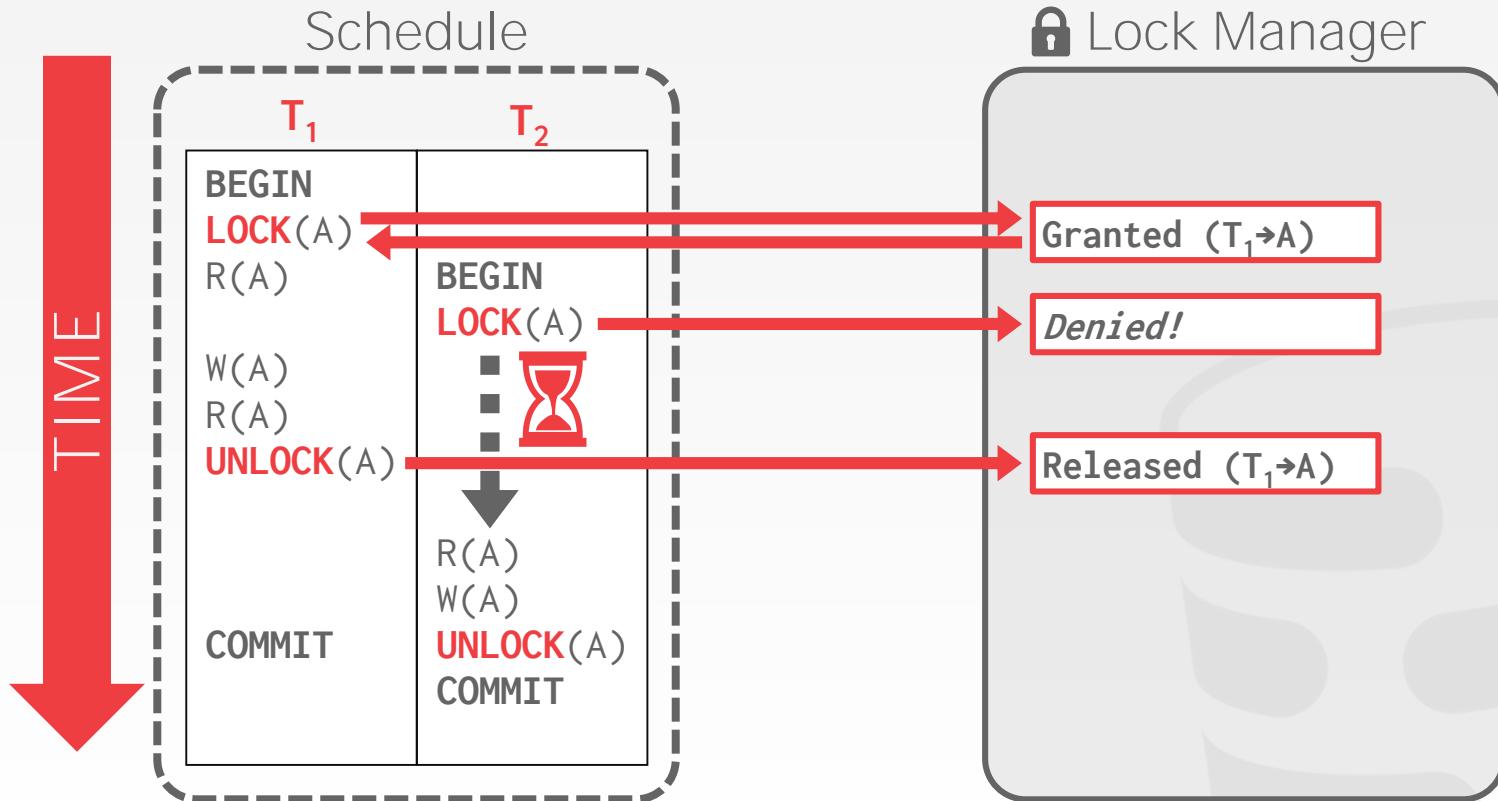
EXECUTING WITH LOCKS



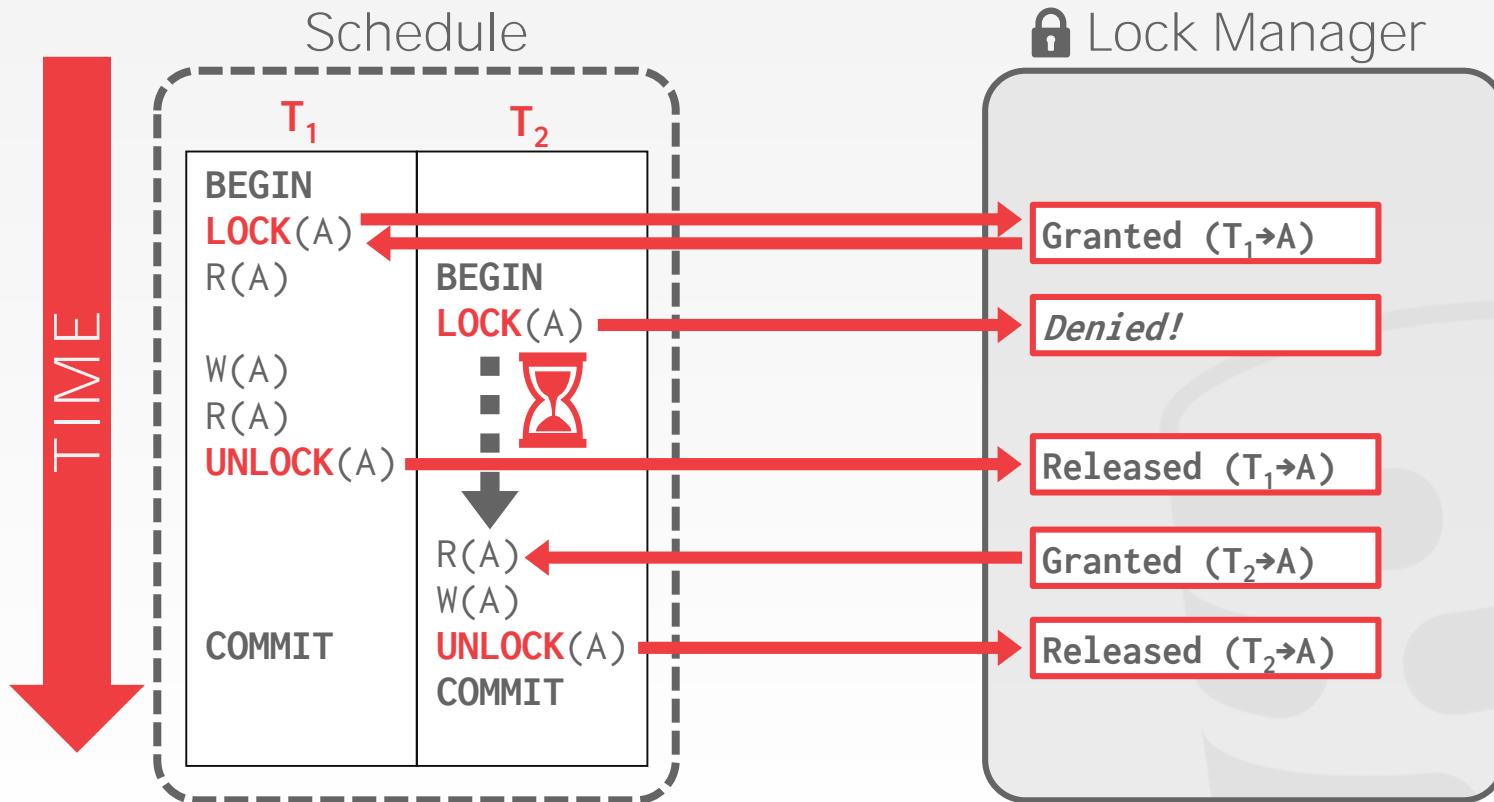
EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



TODAY'S AGENDA

Lock Types

Two-Phase Locking

Deadlock Detection + Prevention

Hierarchical Locking

Isolation Levels



LOCKS VS. LATCHES

Locks

Separate... User transactions

Protect... Database Contents

During... Entire Transactions

Modes... Shared, Exclusive, Update,
Intention

Deadlock Detection & Resolution

...by... Waits-for, Timeout, Aborts

Kept in... Lock Manager

Latches

Threads

In-Memory Data Structures

Critical Sections

Read, Write

Avoidance

Coding Discipline

Protected Data Structure

Source: [Goetz Graefe](#)

BASIC LOCK TYPES

S-LOCK: Shared locks for reads.

X-LOCK: Exclusive locks for writes.

		Compatibility Matrix	
		Shared	Exclusive
Shared	Shared	✓	✗
	Exclusive	✗	✗



EXECUTING WITH LOCKS

Transactions request locks (or upgrades).

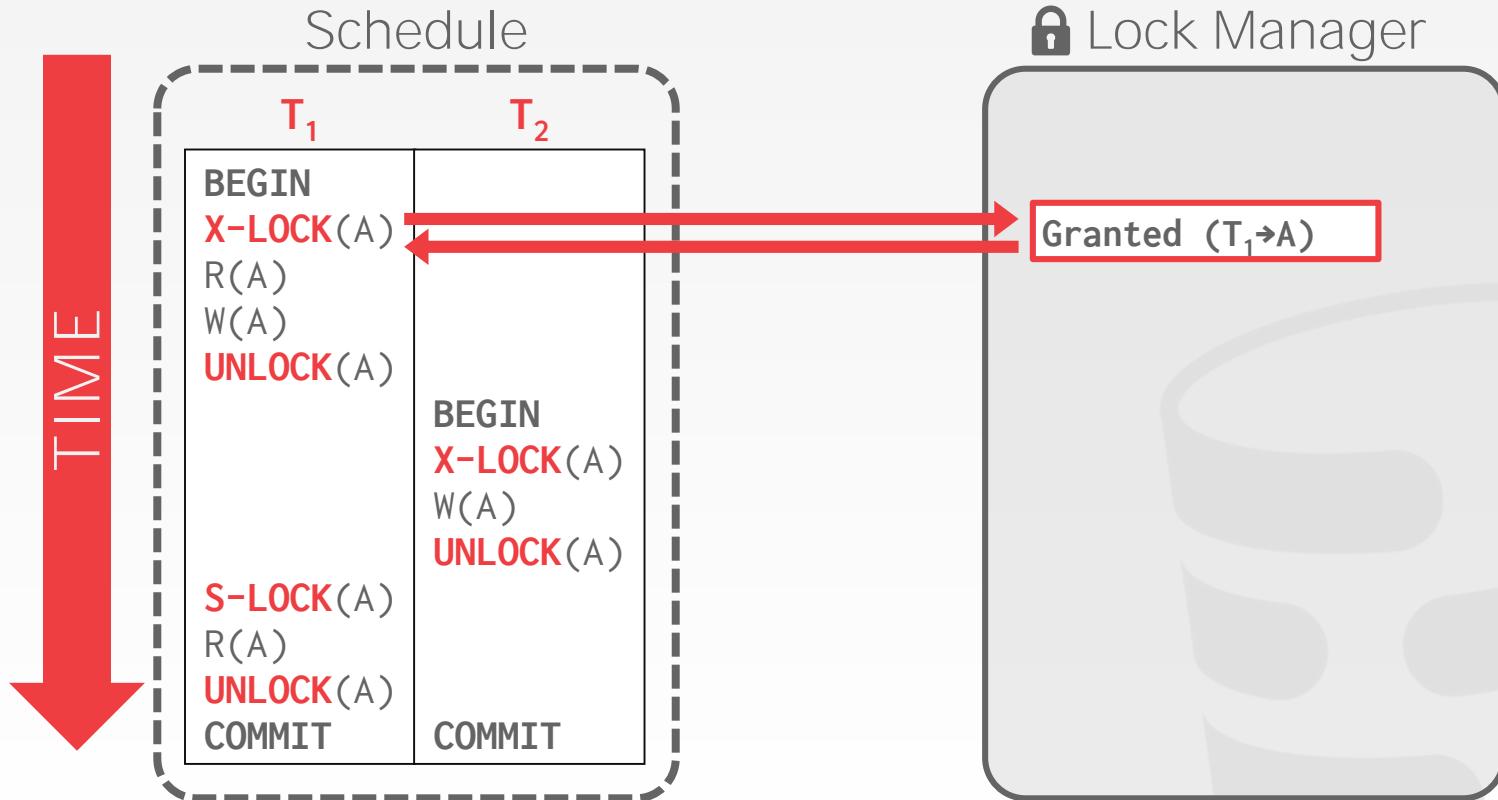
Lock manager grants or blocks requests.

Transactions release locks.

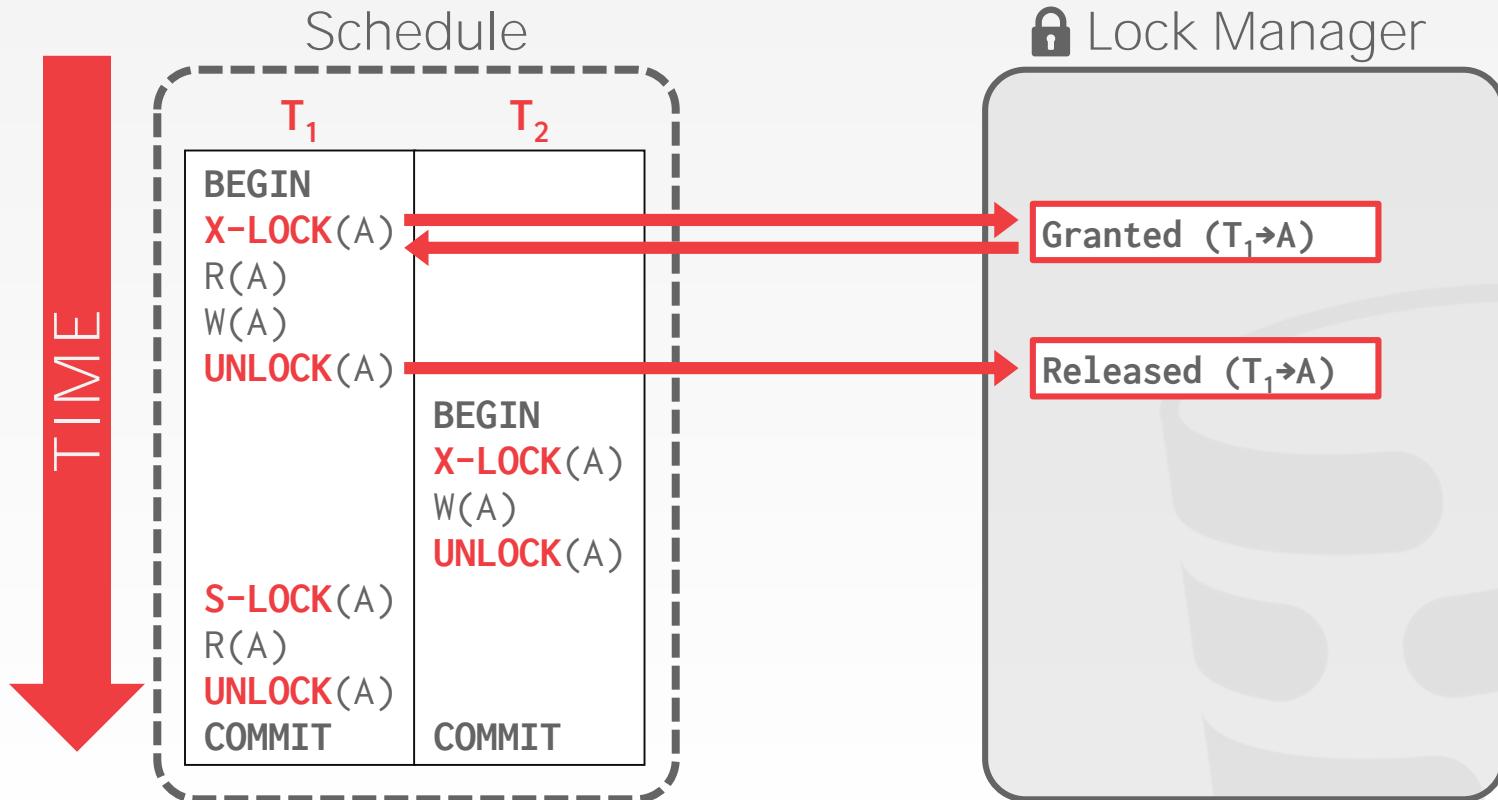
Lock manager updates its internal lock-table.

→ It keeps track of what transactions hold what locks and what transactions are waiting to acquire any locks.

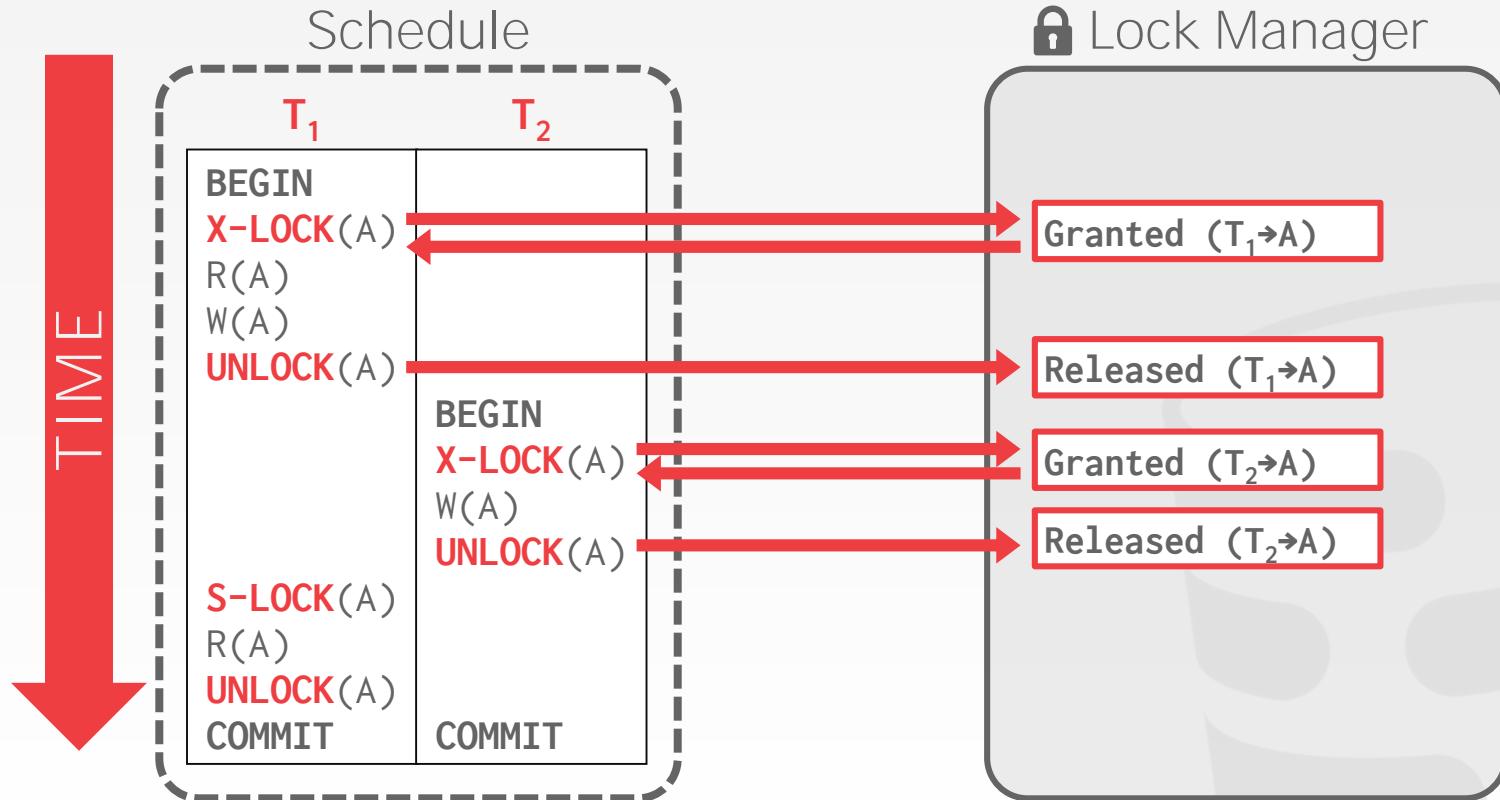
EXECUTING WITH LOCKS



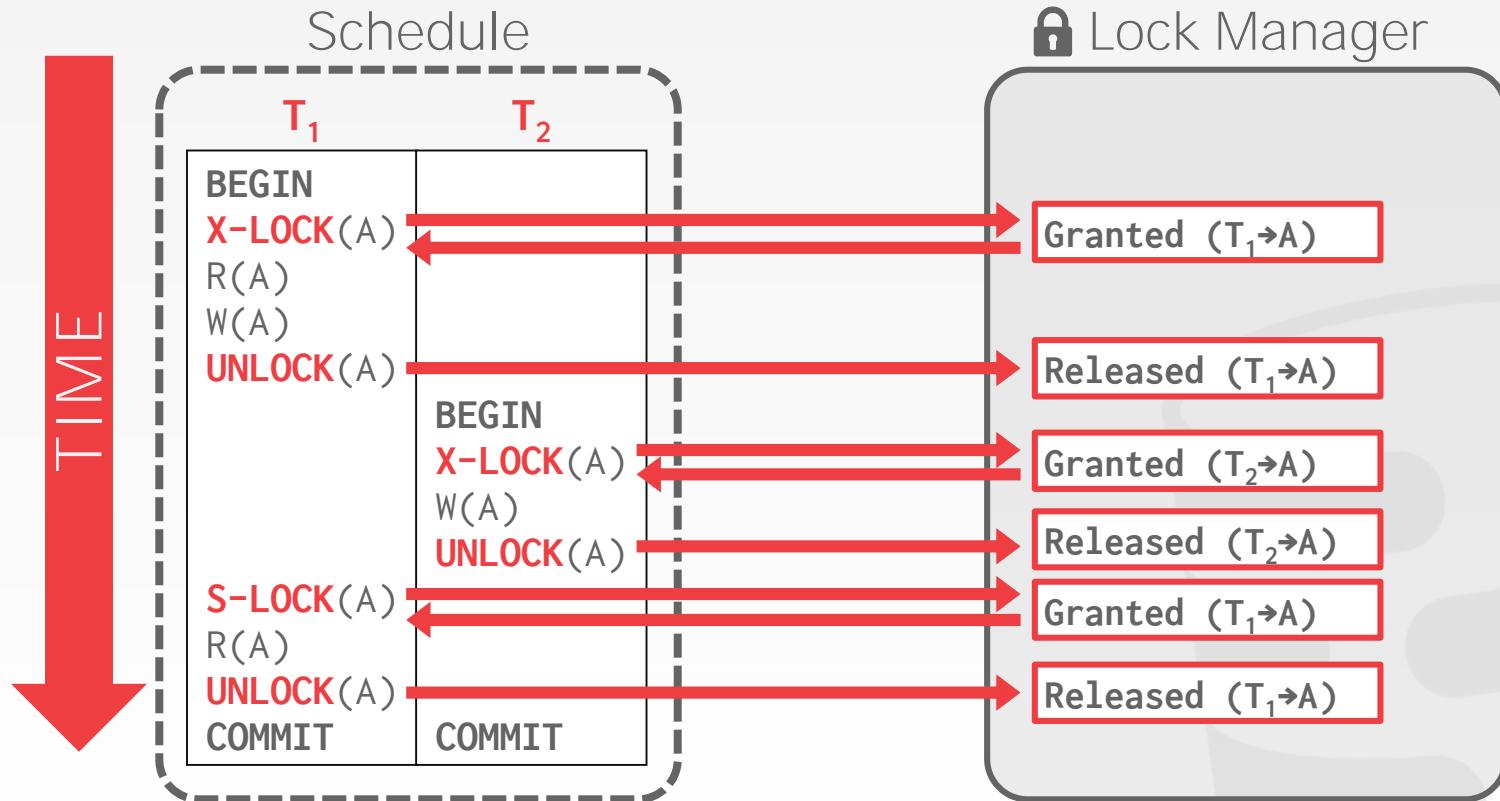
EXECUTING WITH LOCKS



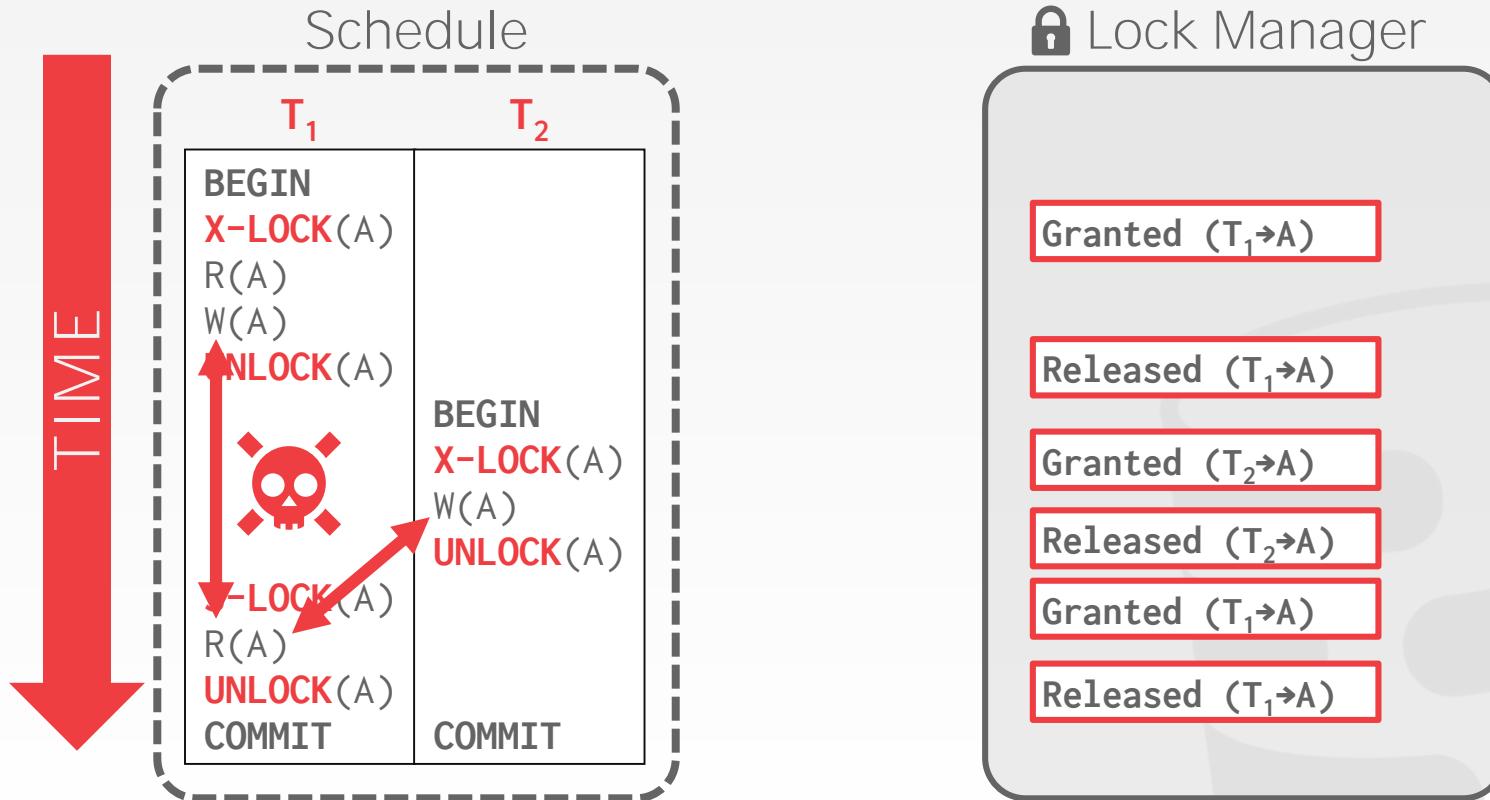
EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



CONCURRENCY CONTROL PROTOCOL

Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn can access an object in the database on the fly.

The protocol does not need to know all the queries that a txn will execute ahead of time.



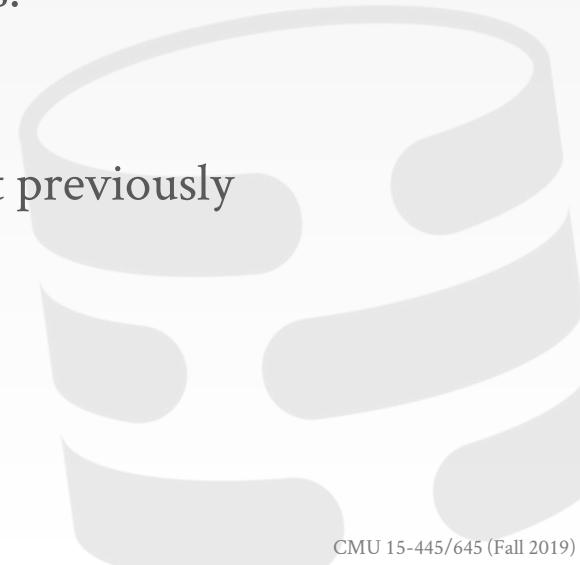
TWO-PHASE LOCKING

Phase #1: Growing

- Each txn requests the locks that it needs from the DBMS's lock manager.
- The lock manager grants/denies lock requests.

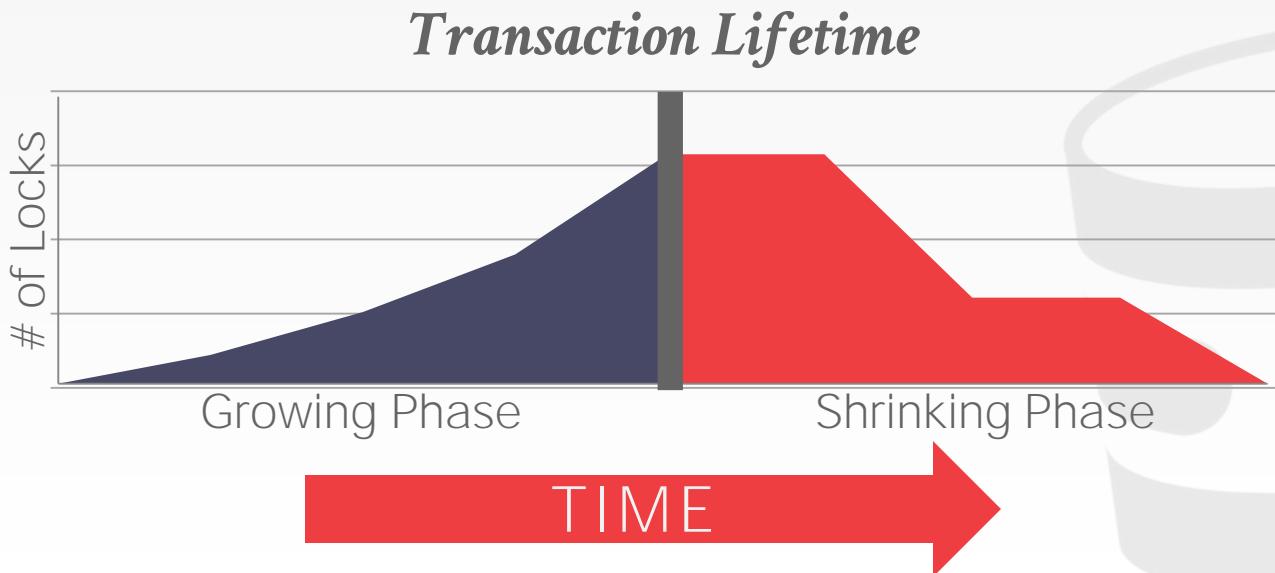
Phase #2: Shrinking

- The txn is allowed to only release locks that it previously acquired. It cannot acquire new locks.



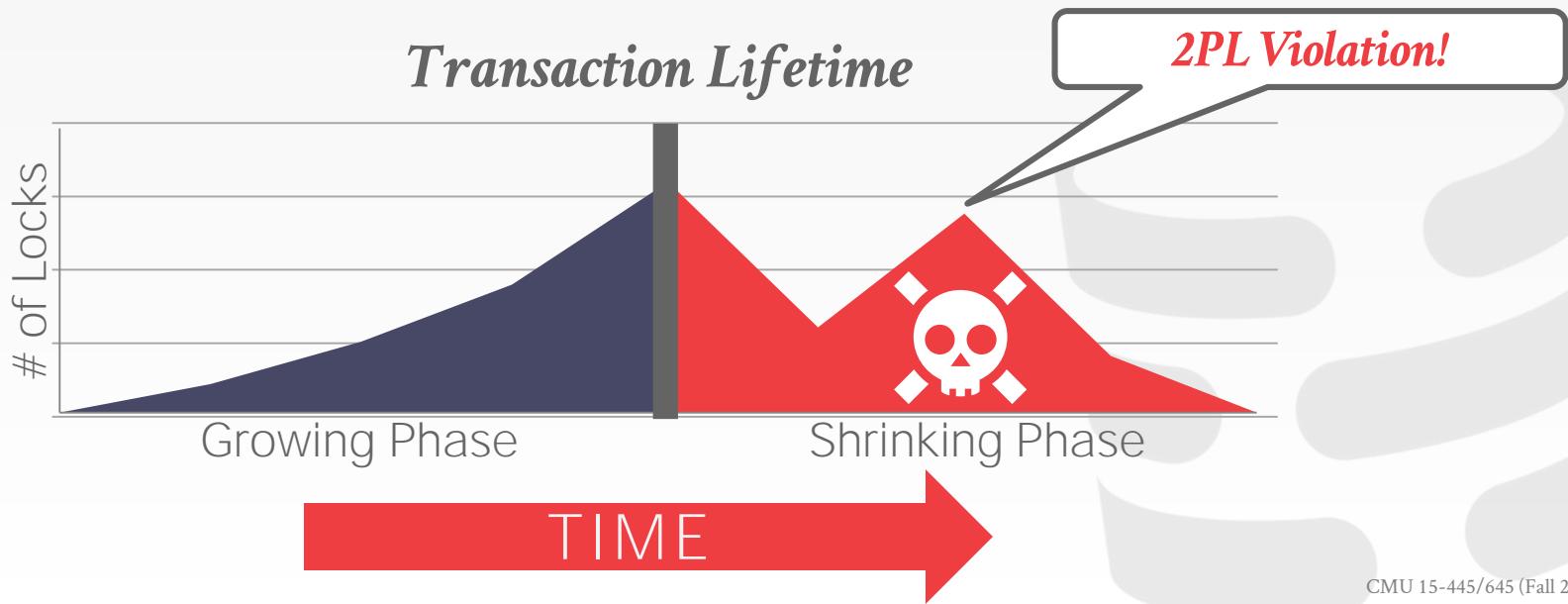
TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

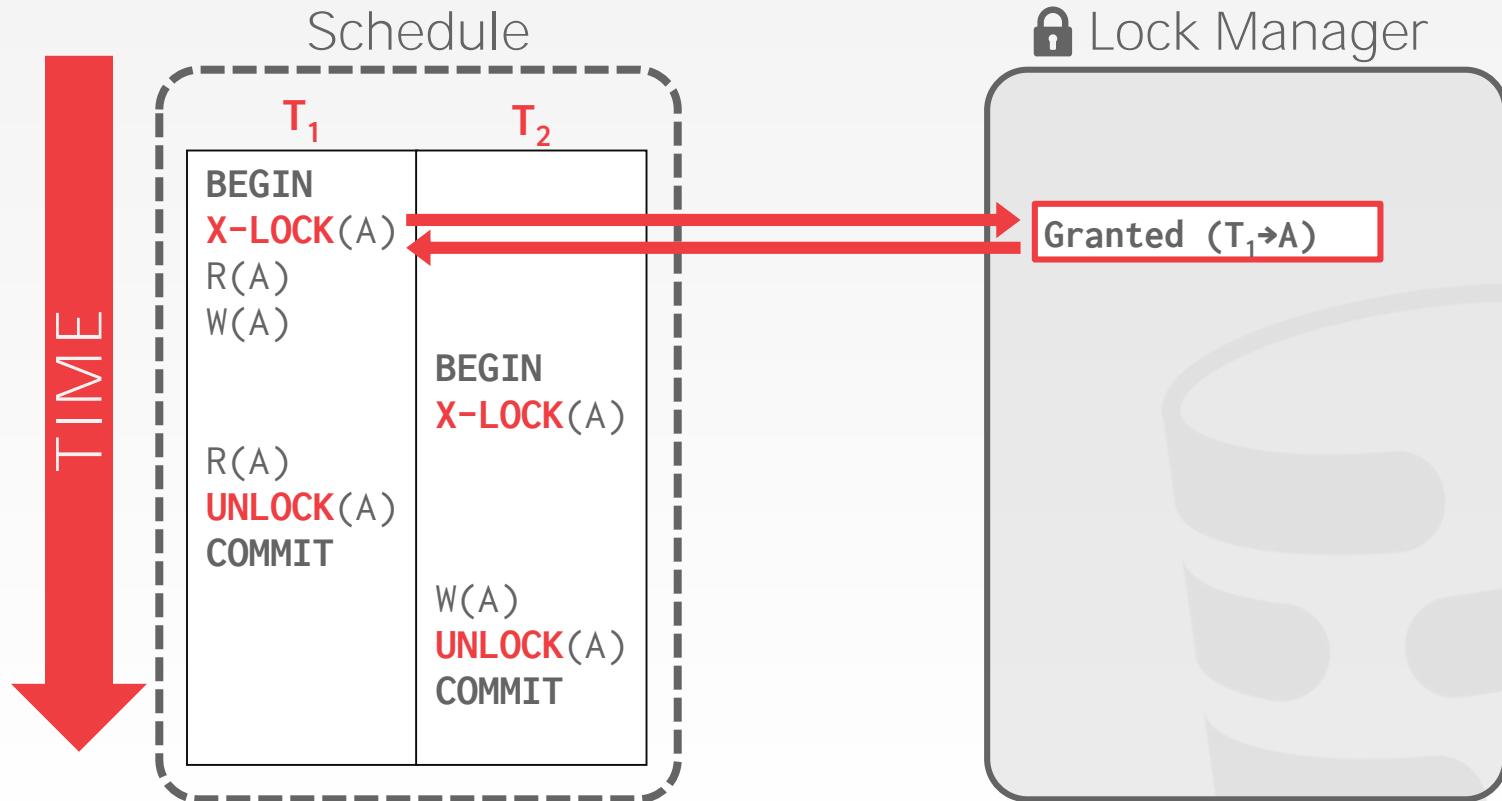


TWO-PHASE LOCKING

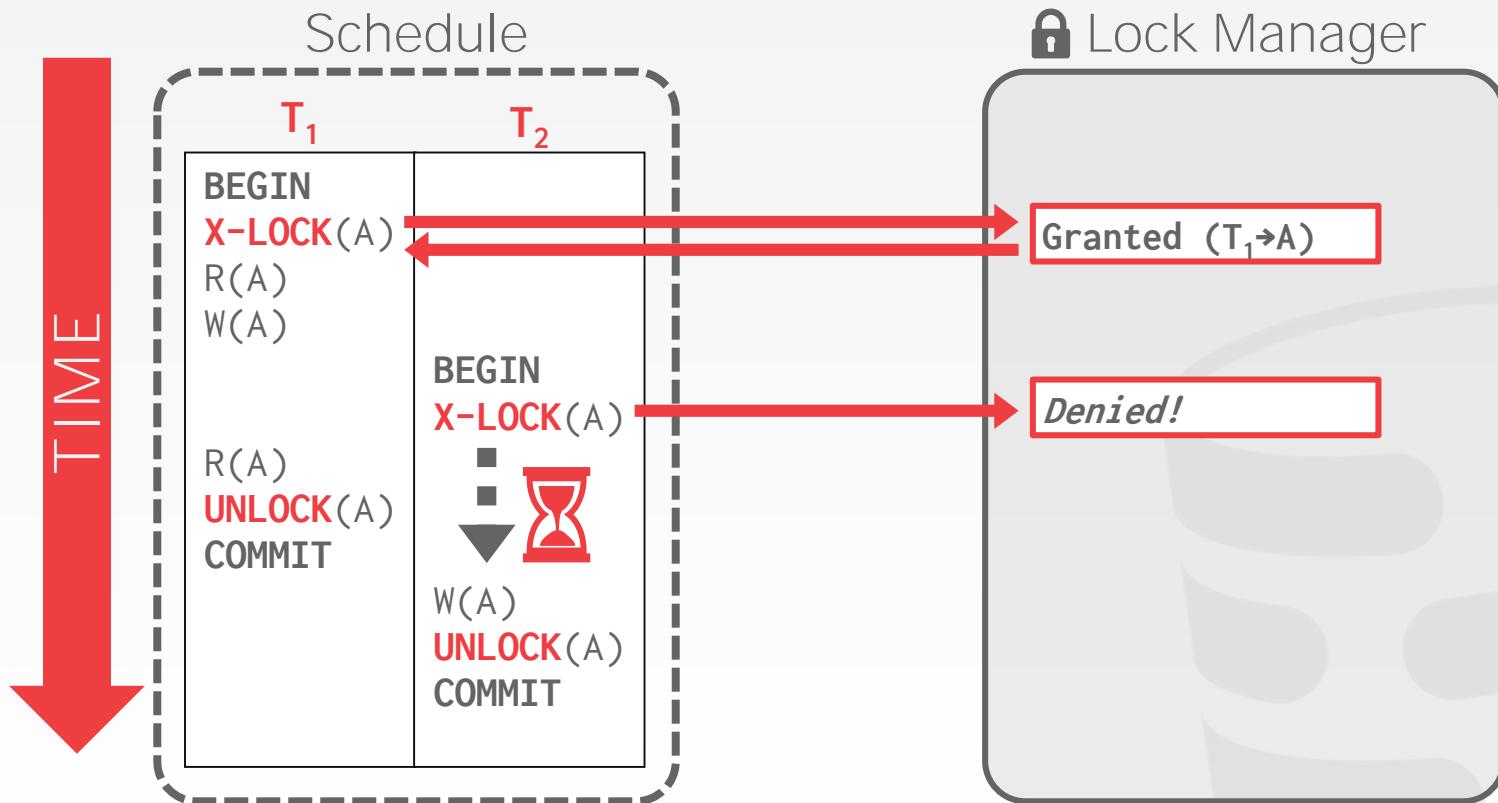
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.



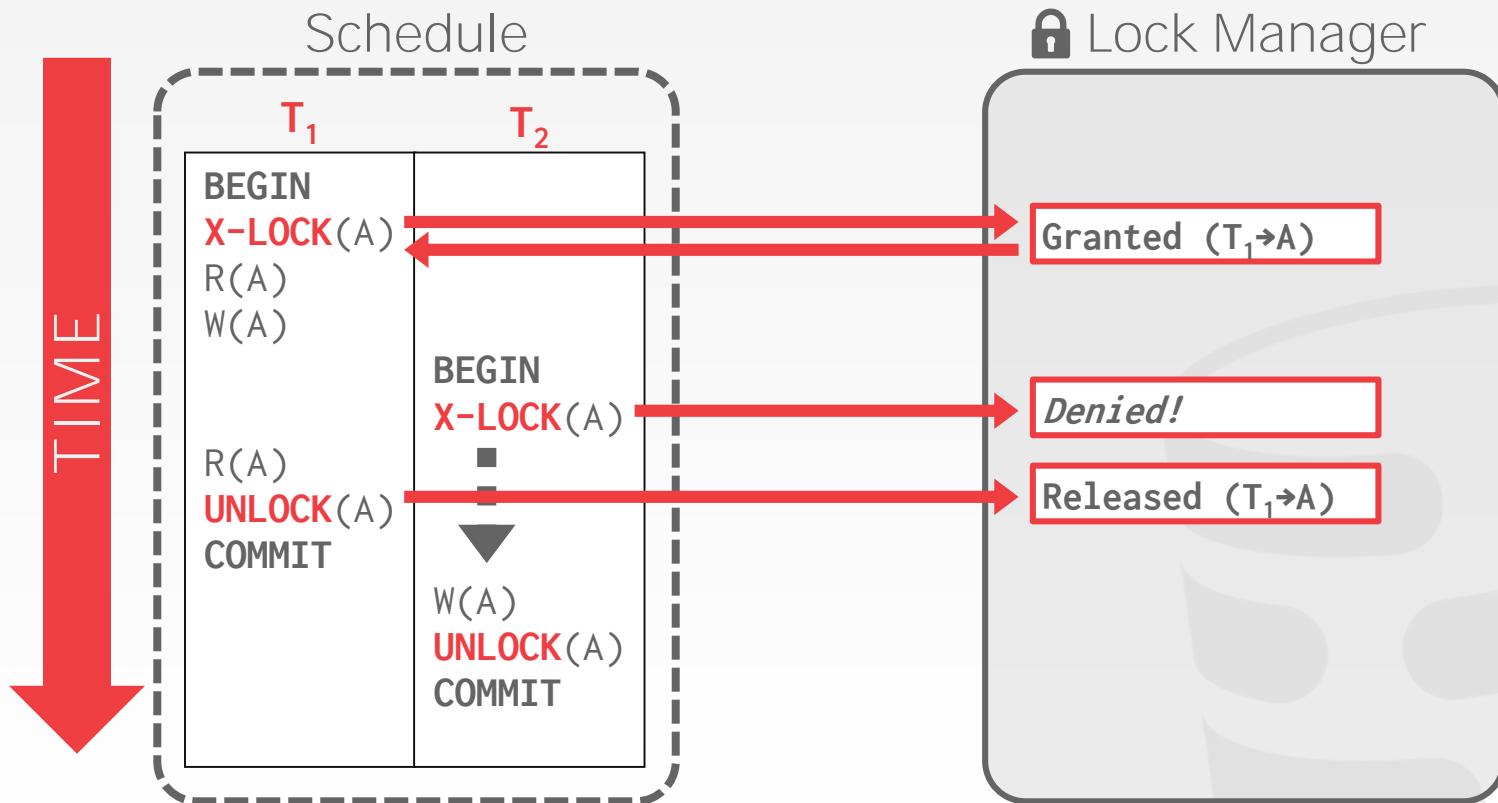
EXECUTING WITH 2PL



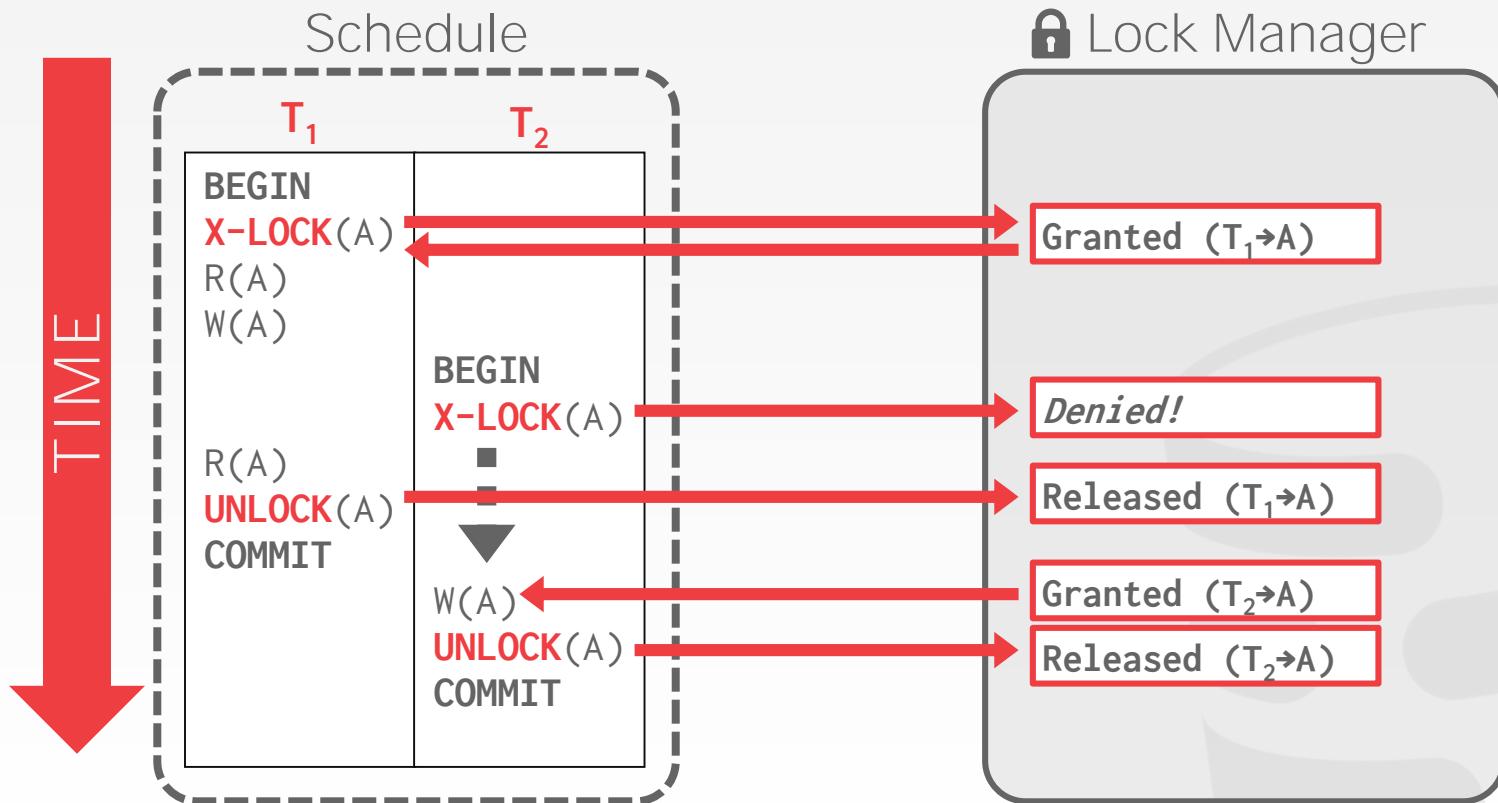
EXECUTING WITH 2PL



EXECUTING WITH 2PL



EXECUTING WITH 2PL



TWO-PHASE LOCKING

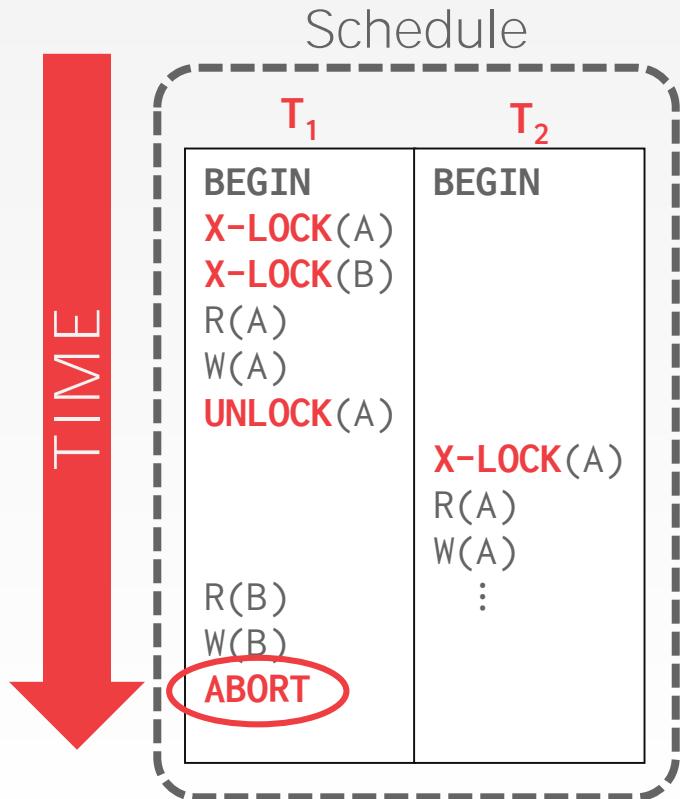
2PL on its own is sufficient to guarantee conflict serializability.

→ It generates schedules whose precedence graph is acyclic.

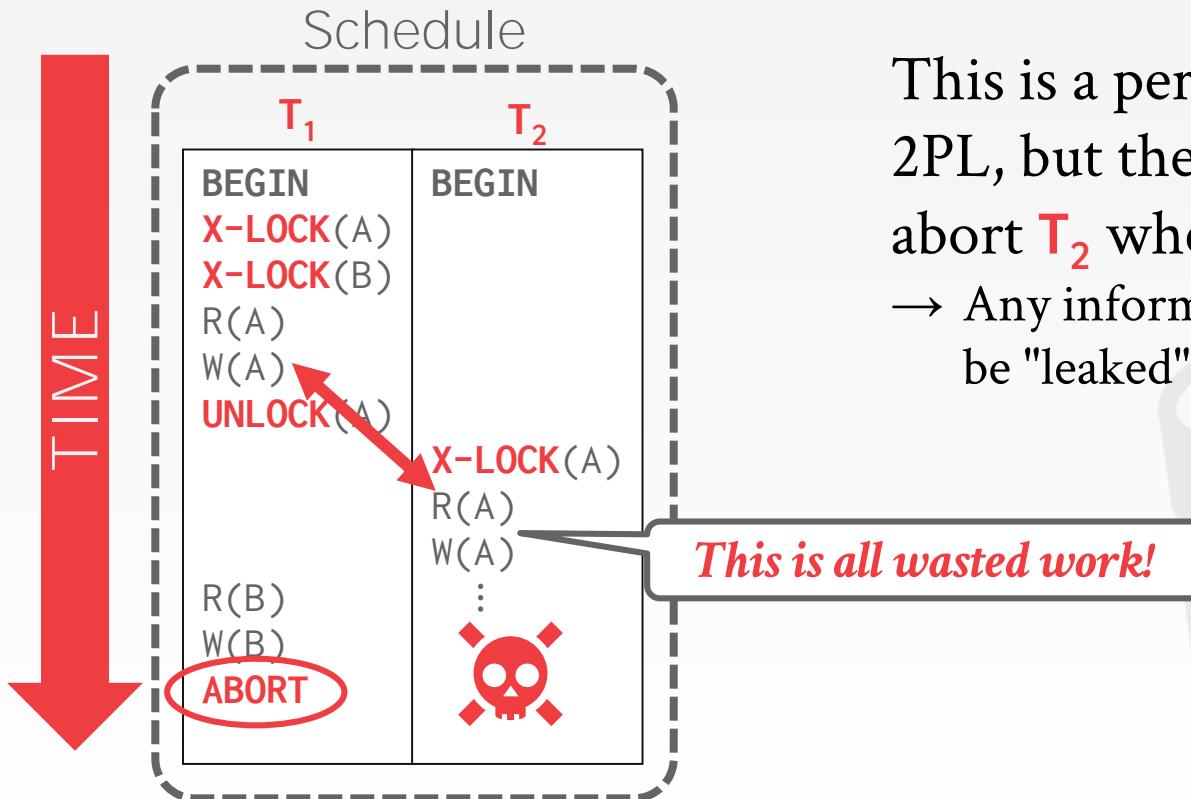
But it is subject to cascading aborts.



2PL – CASCADING ABORTS



2PL – CASCADING ABORTS



This is a permissible schedule in 2PL, but the DBMS has to also abort T_2 when T_1 aborts.
 → Any information about T_1 cannot be "leaked" to the outside world.

2PL OBSERVATIONS

There are potential schedules that are serializable but would not be allowed by 2PL.
→ Locking limits concurrency.

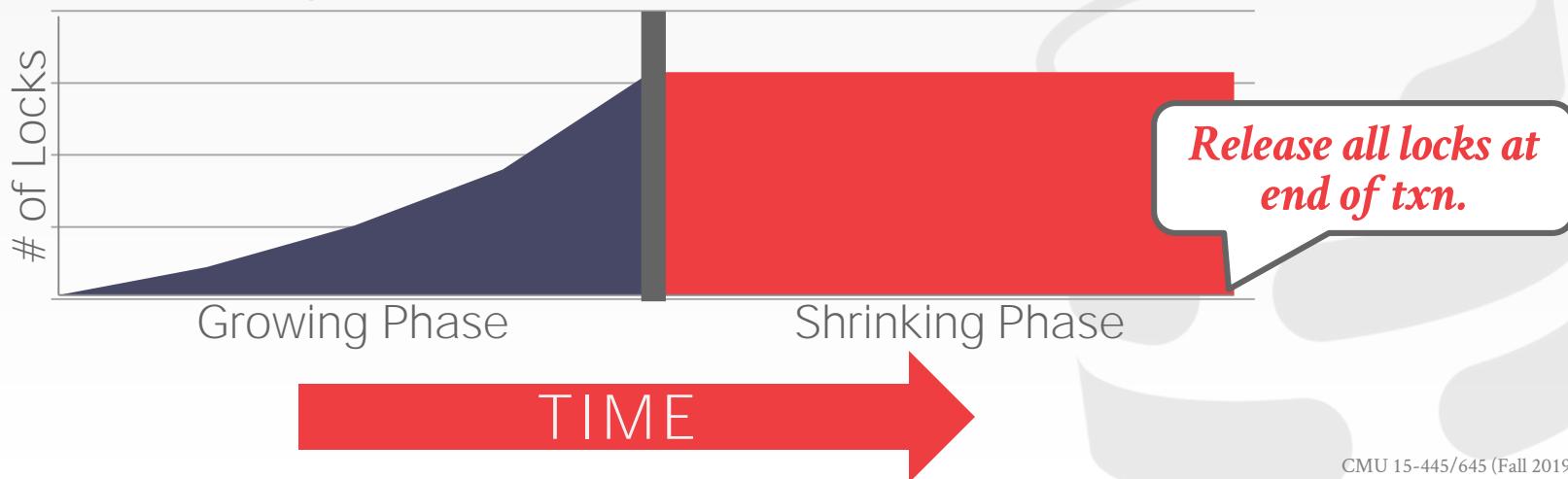
May still have "dirty reads".
→ Solution: **Strong Strict 2PL (aka Rigorous 2PL)**

May lead to deadlocks.
→ Solution: **Detection or Prevention**

STRONG STRICT TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

Allows only conflict serializable schedules, but it is often stronger than needed for some apps.



STRONG STRICT TWO-PHASE LOCKING

A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes.

Advantages:

- Does not incur cascading aborts.
- Aborted txns can be undone by just restoring original values of modified tuples.



EXAMPLES

T₁ – Move \$100 from Andy's account (**A**) to his bookie's account (**B**).

T₂ – Compute the total amount in all accounts and return it to the application.

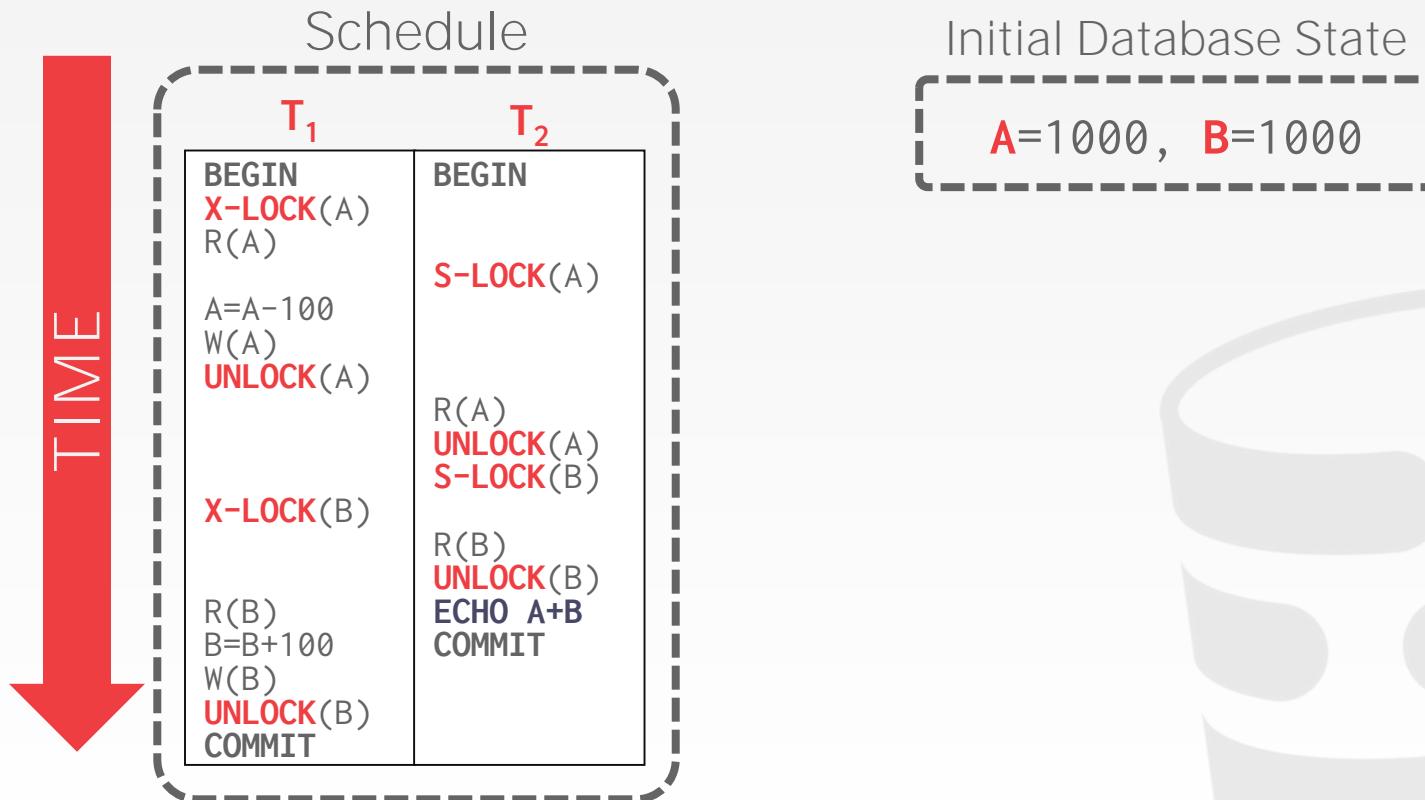
T₁

```
BEGIN  
A=A-100  
B=B+100  
COMMIT
```

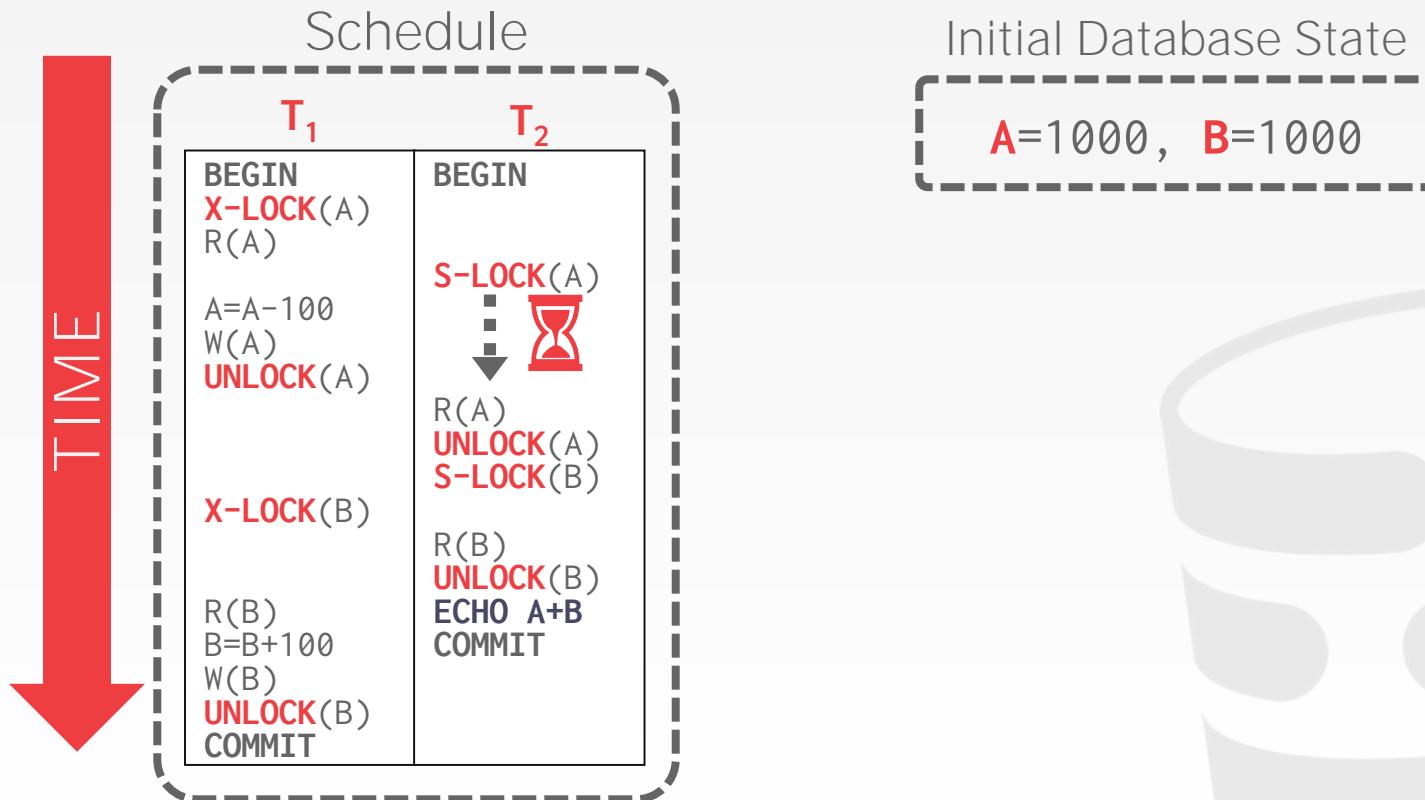
T₂

```
BEGIN  
ECHO A+B  
COMMIT
```

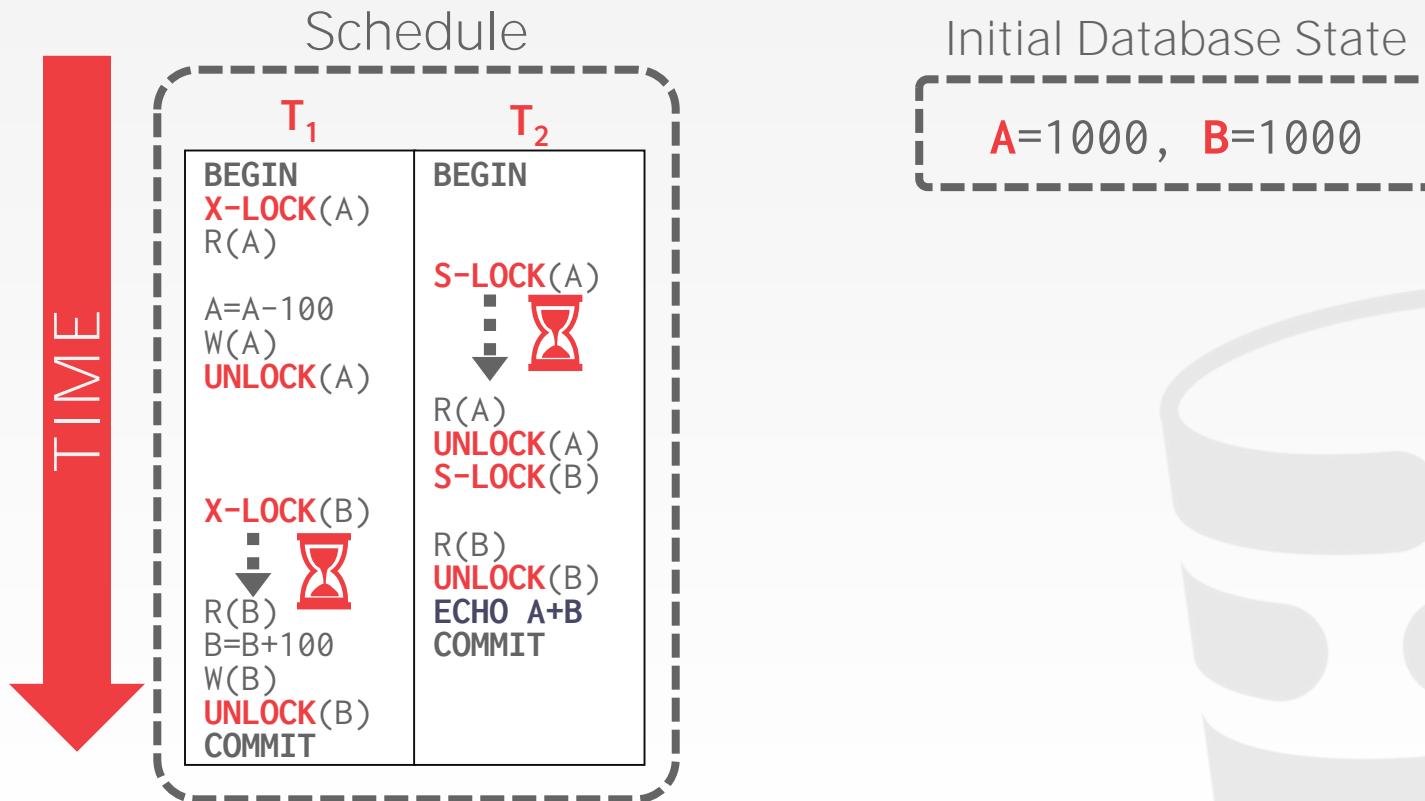
NON-2PL EXAMPLE



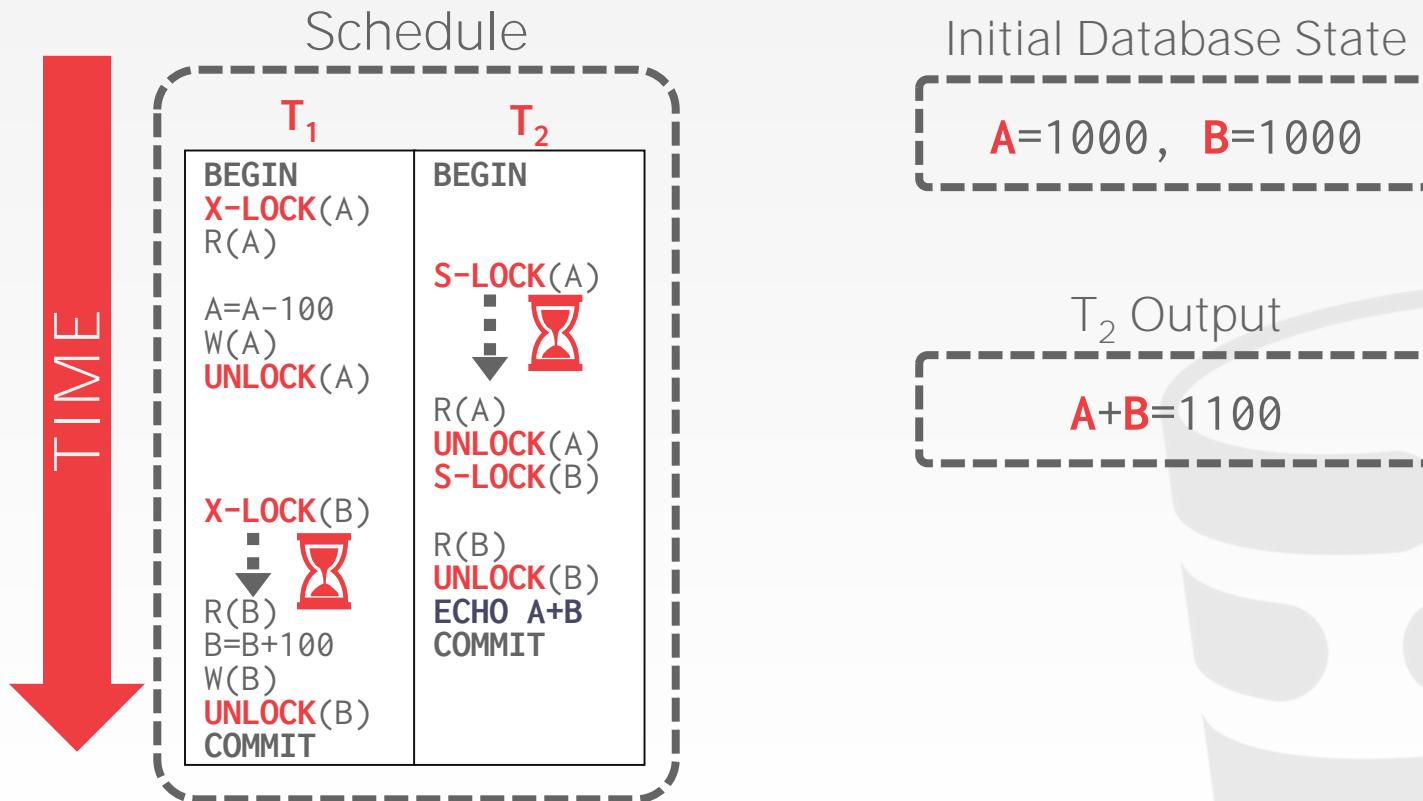
NON-2PL EXAMPLE



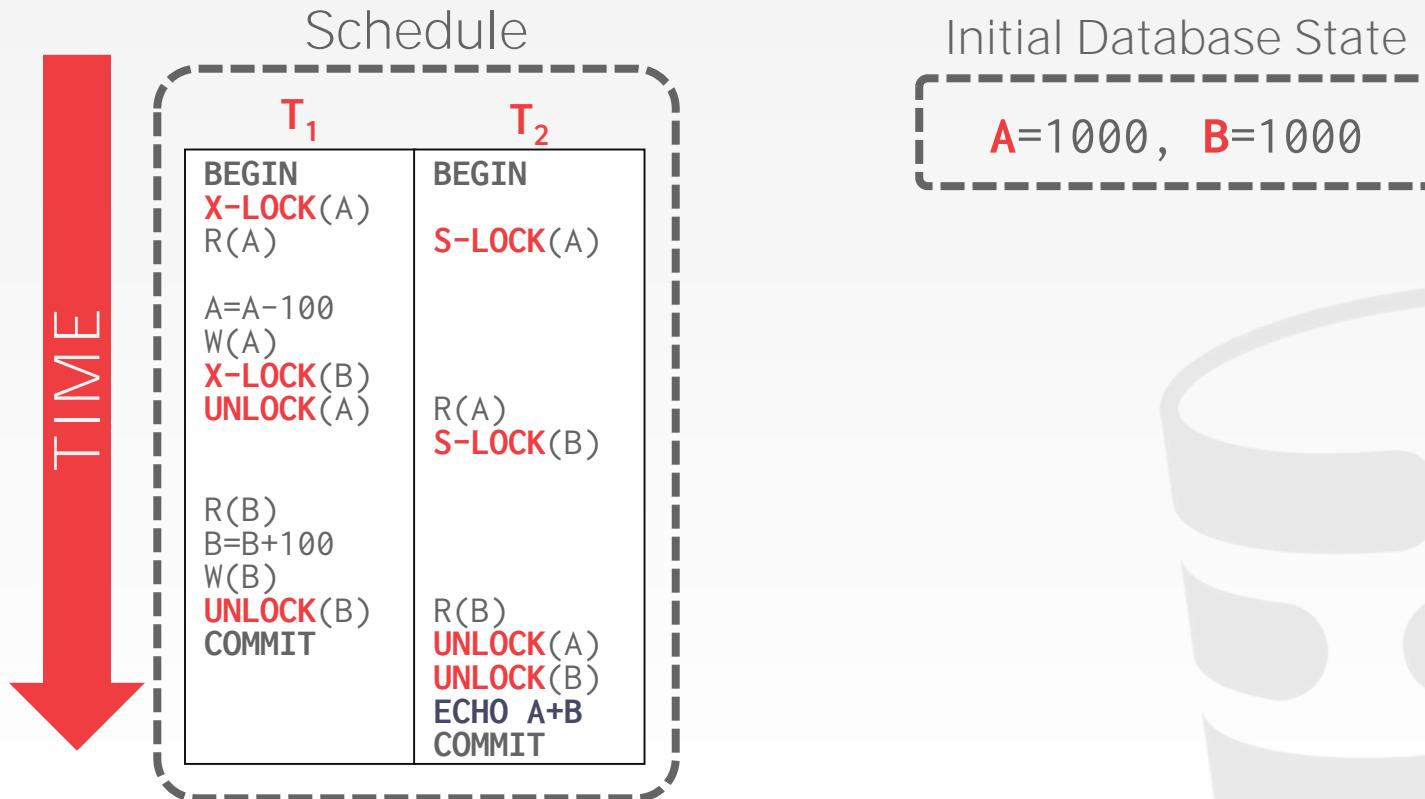
NON-2PL EXAMPLE



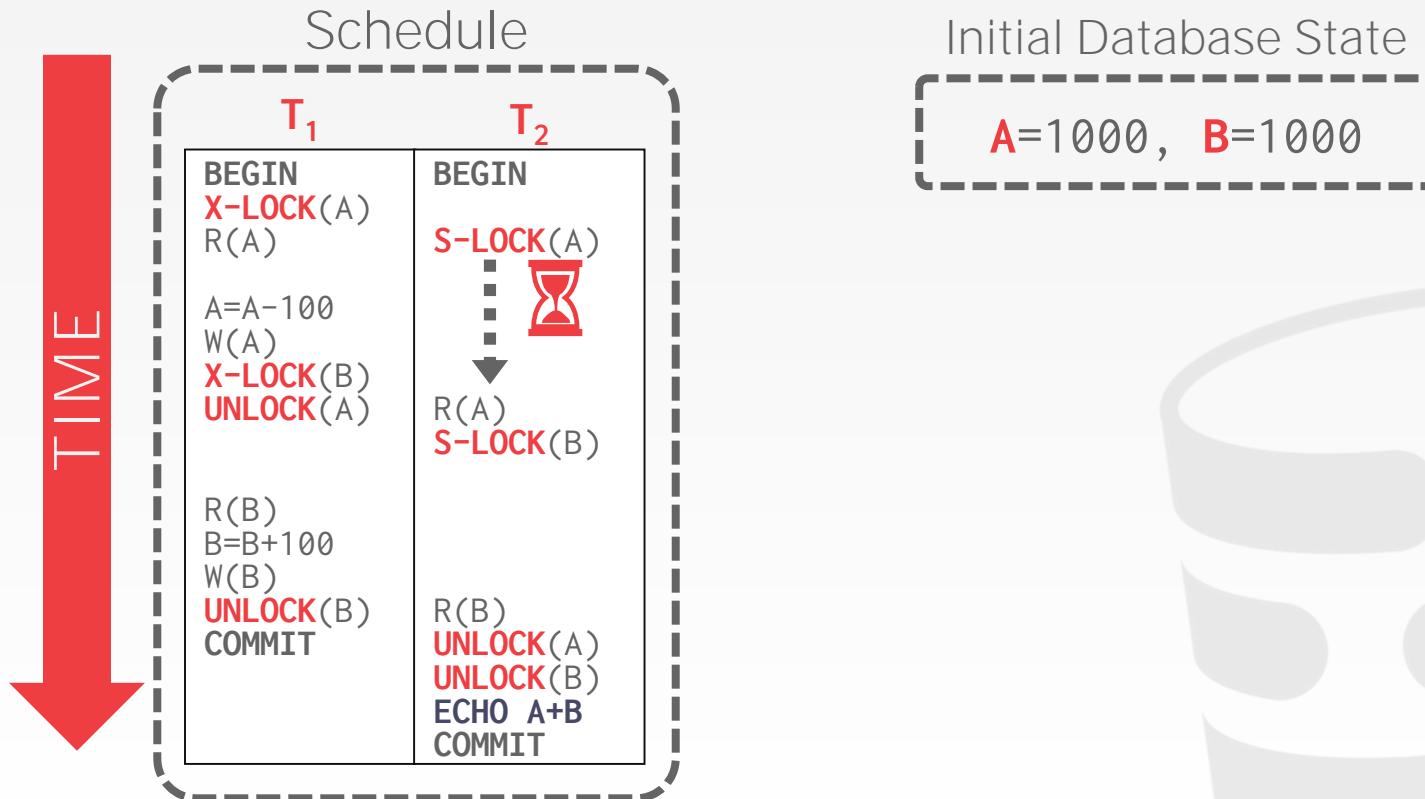
NON-2PL EXAMPLE



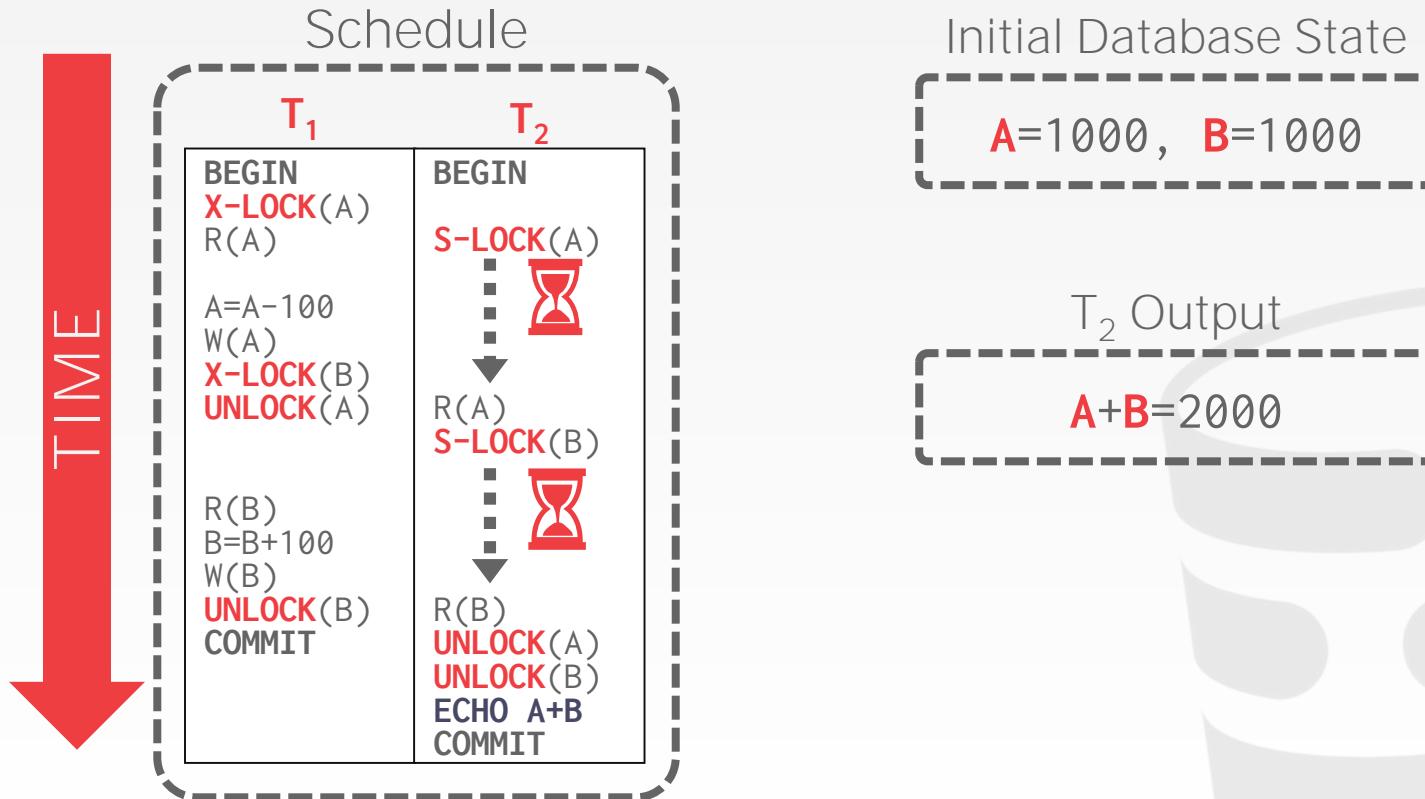
2PL EXAMPLE



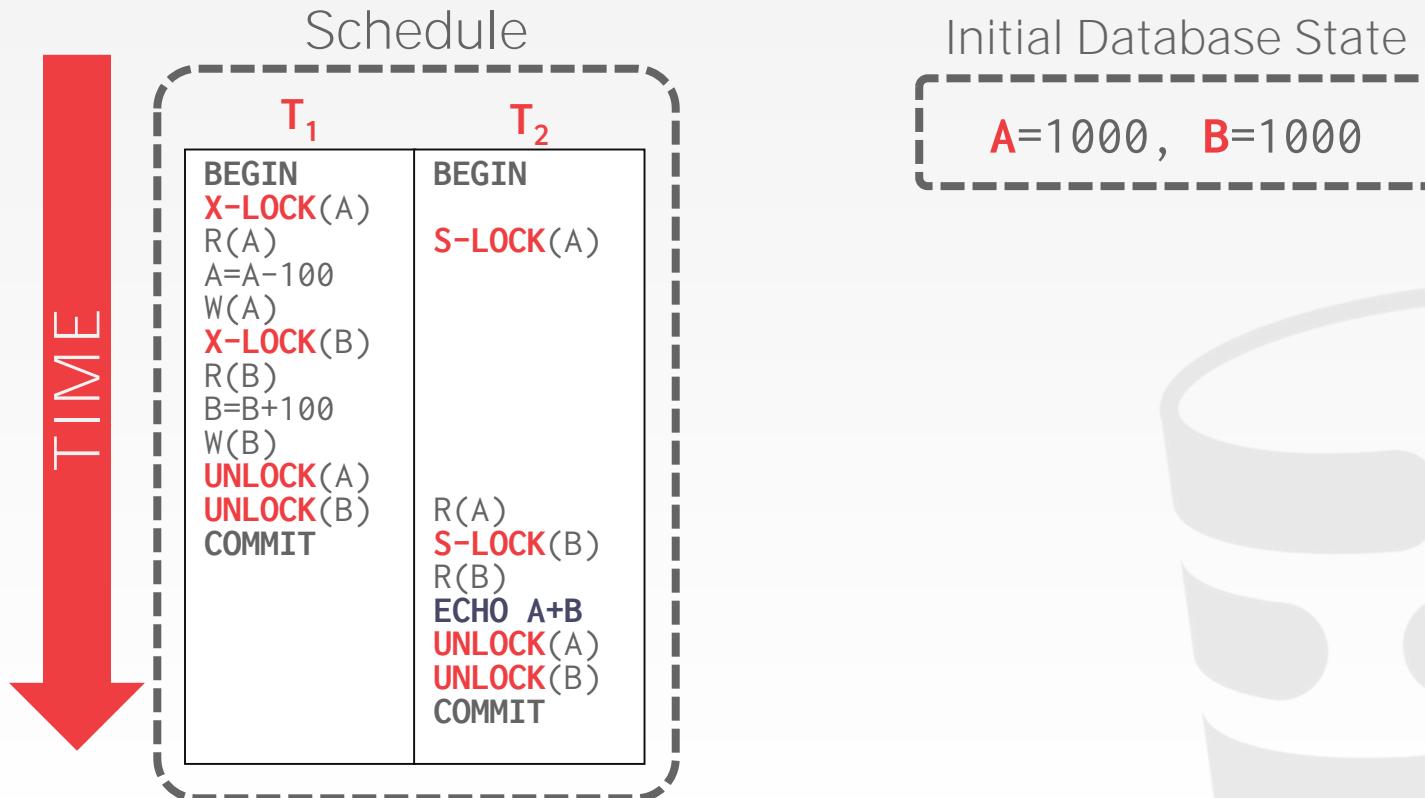
2PL EXAMPLE



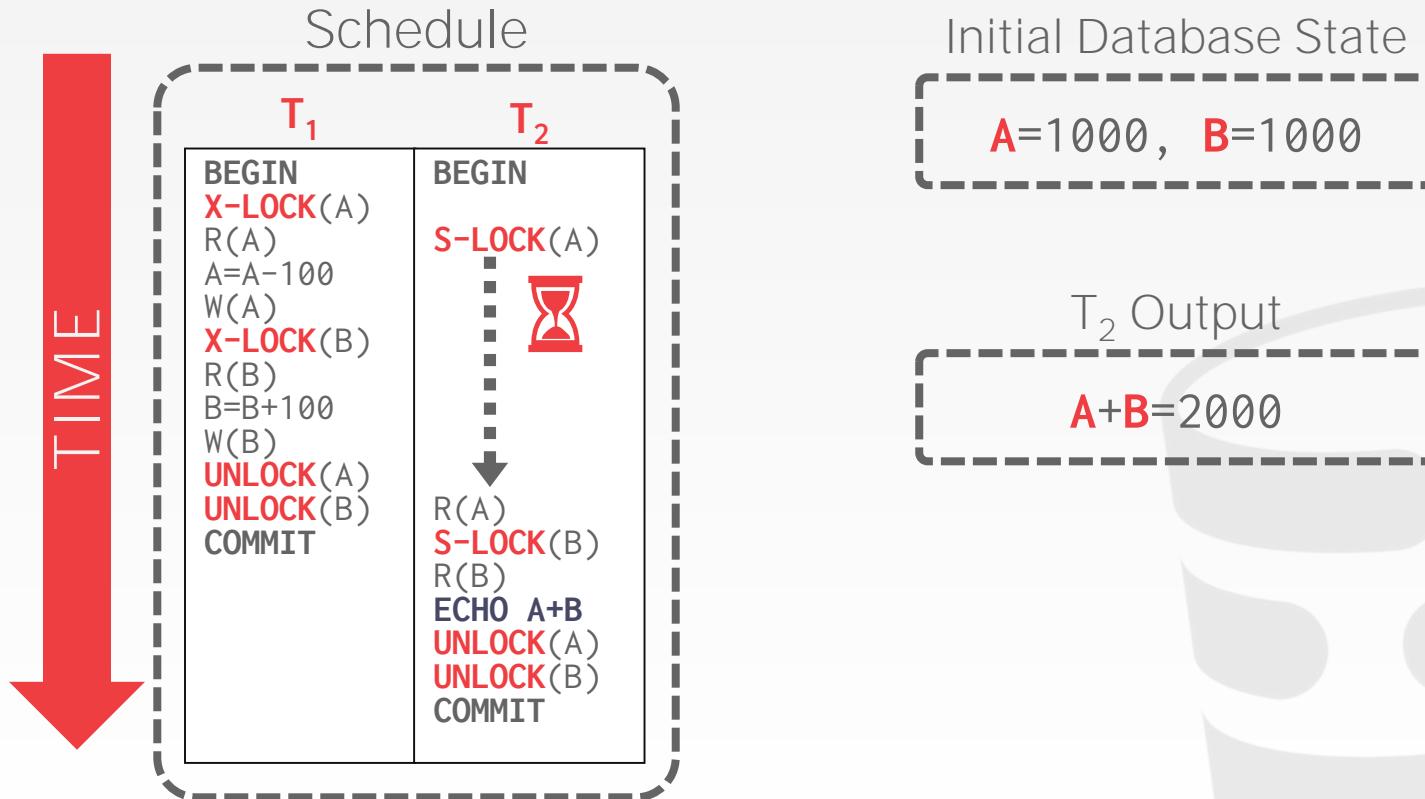
2PL EXAMPLE



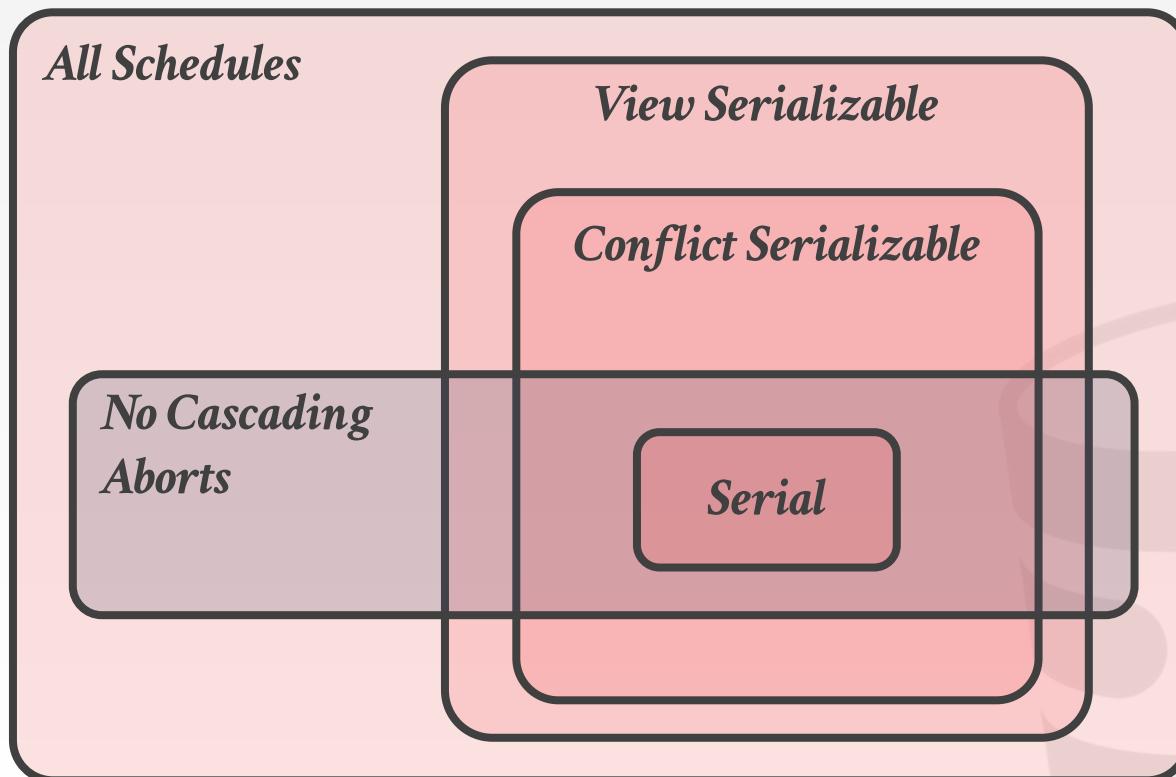
STRONG STRICT 2PL EXAMPLE



STRONG STRICT 2PL EXAMPLE



UNIVERSE OF SCHEDULES



2PL OBSERVATIONS

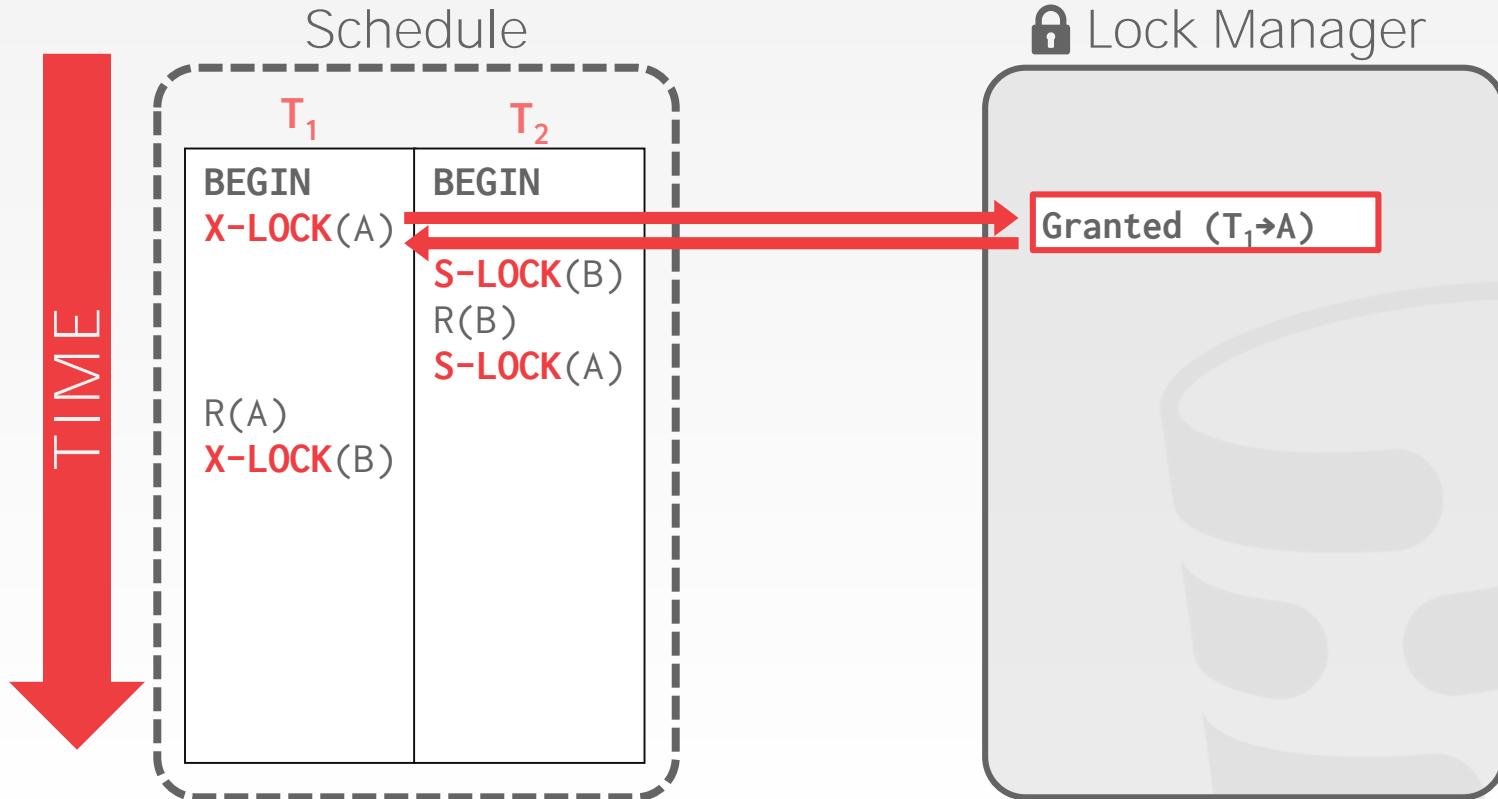
There are potential schedules that are serializable but would not be allowed by 2PL.
→ Locking limits concurrency.

May still have "dirty reads".
→ Solution: **Strong Strict 2PL (Rigorous)**

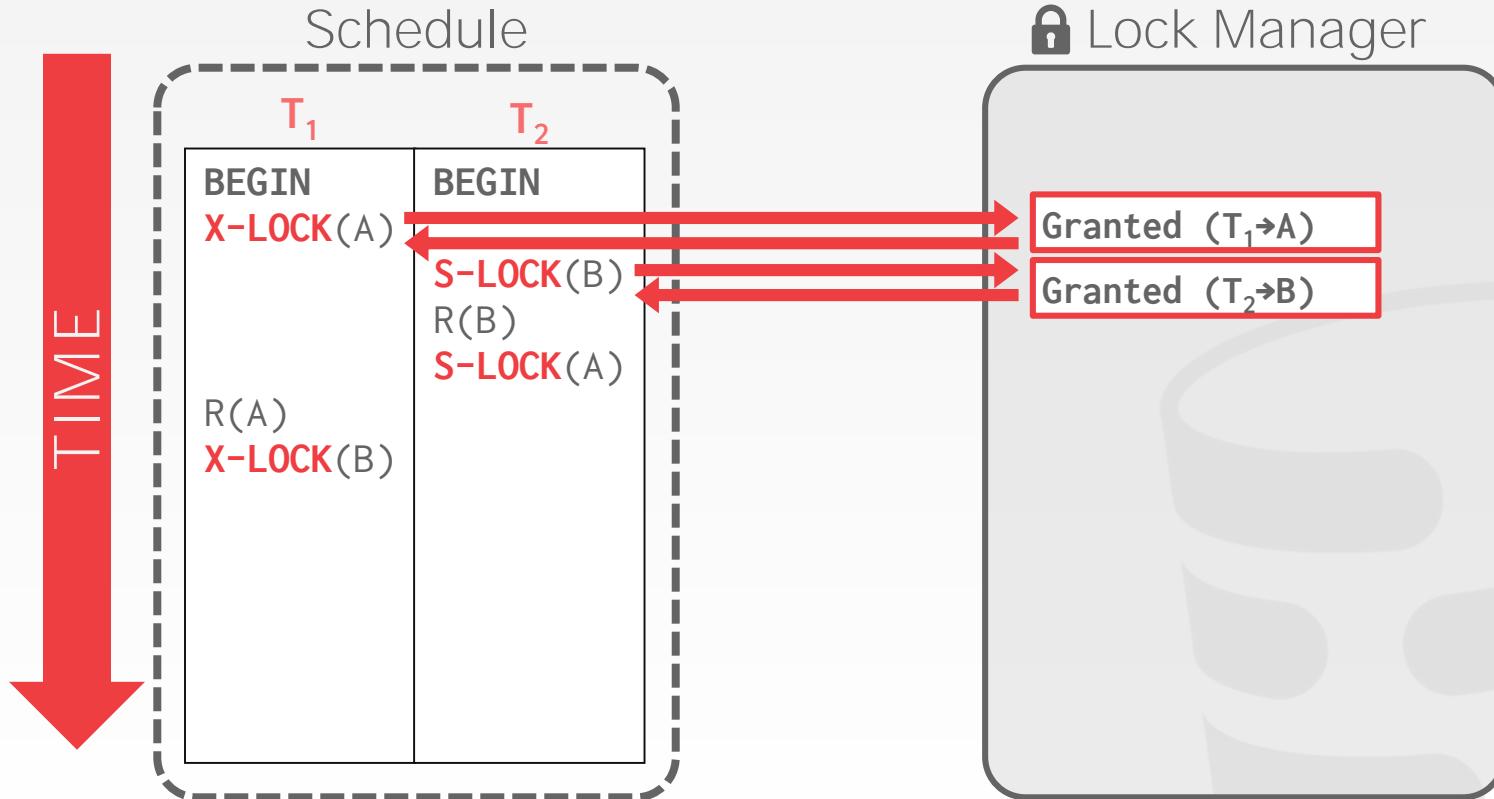
May lead to deadlocks.
→ Solution: **Detection or Prevention**



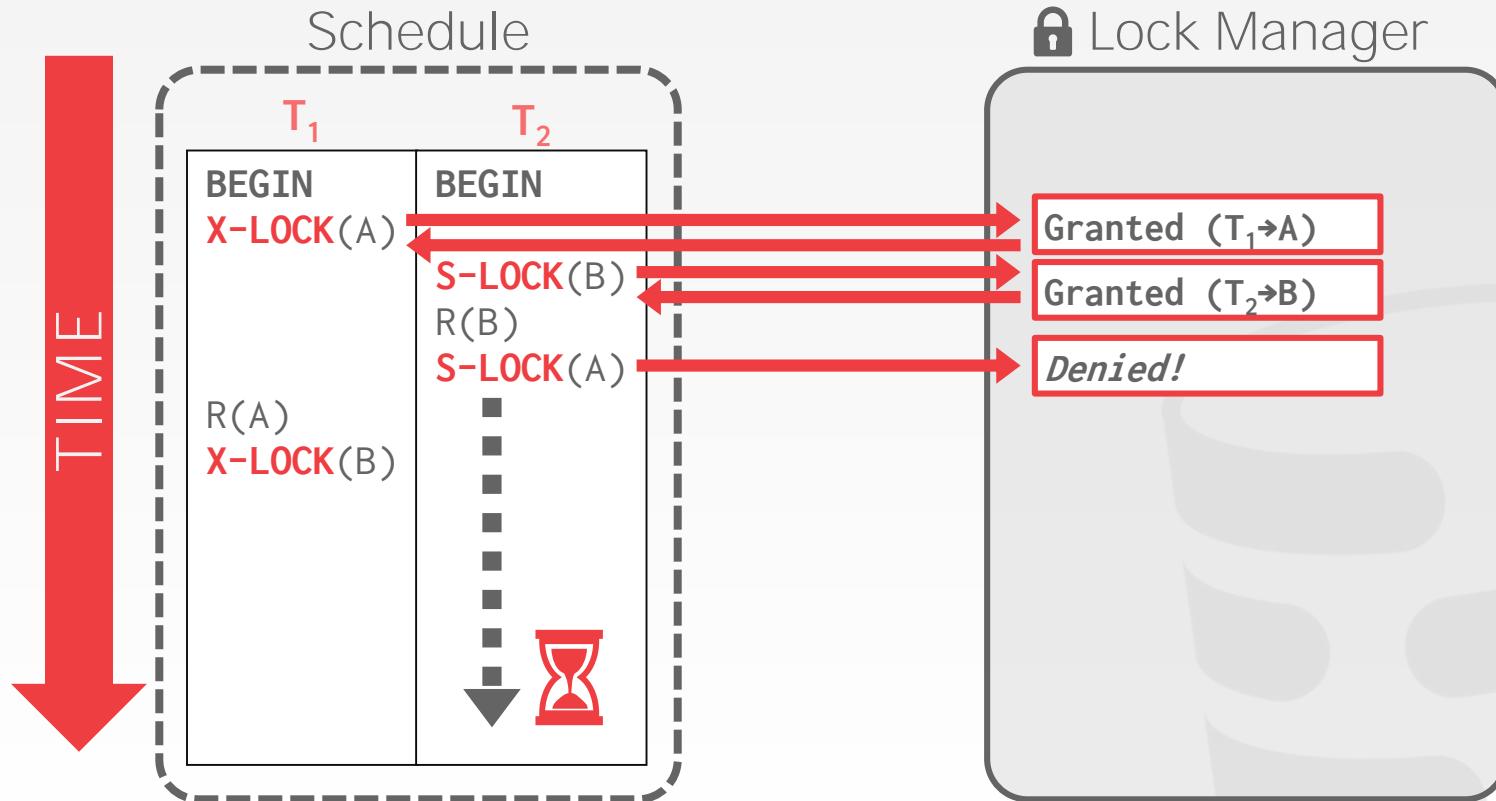
SHIT JUST GOT REAL, SON



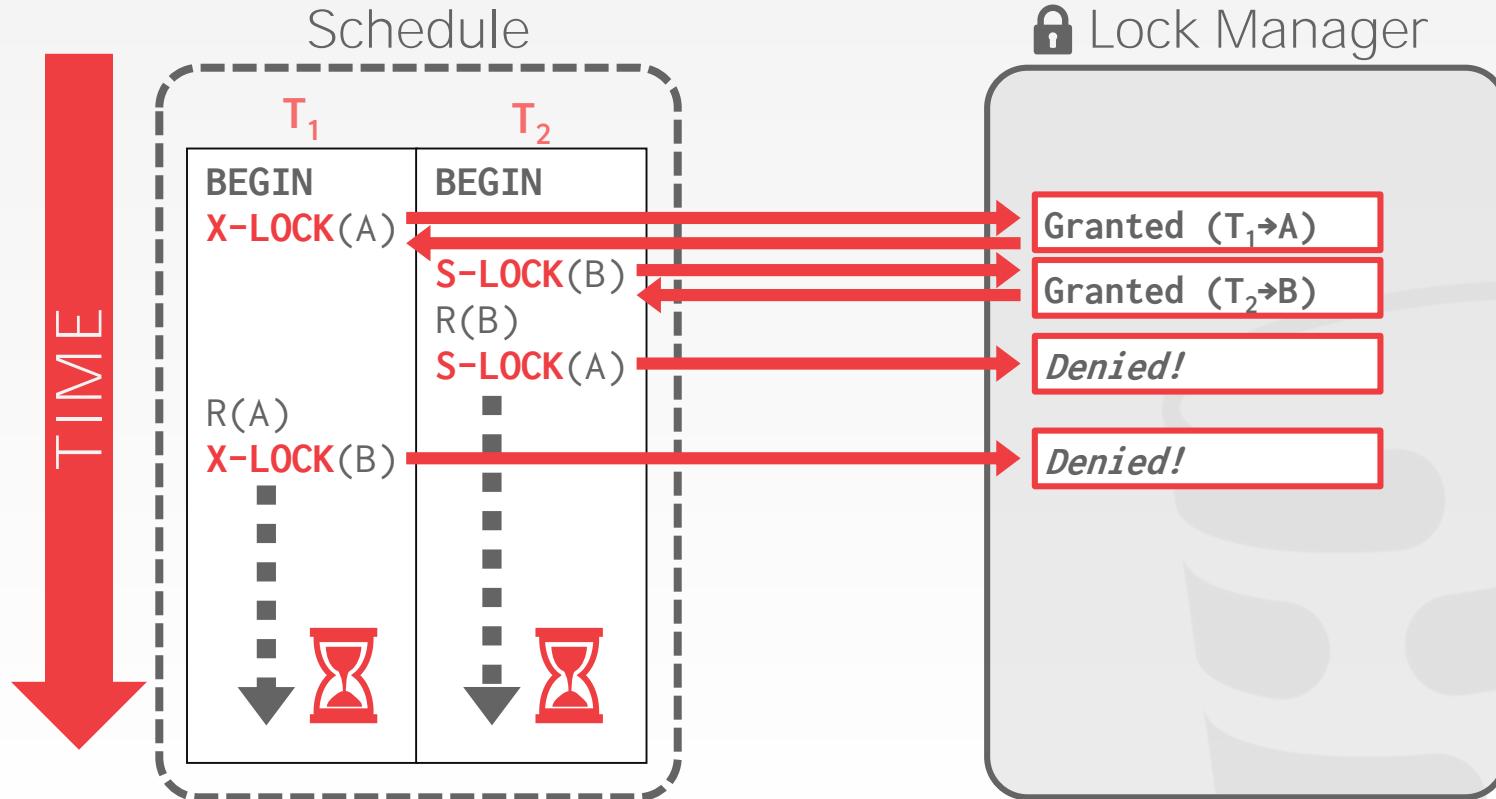
SHIT JUST GOT REAL, SON



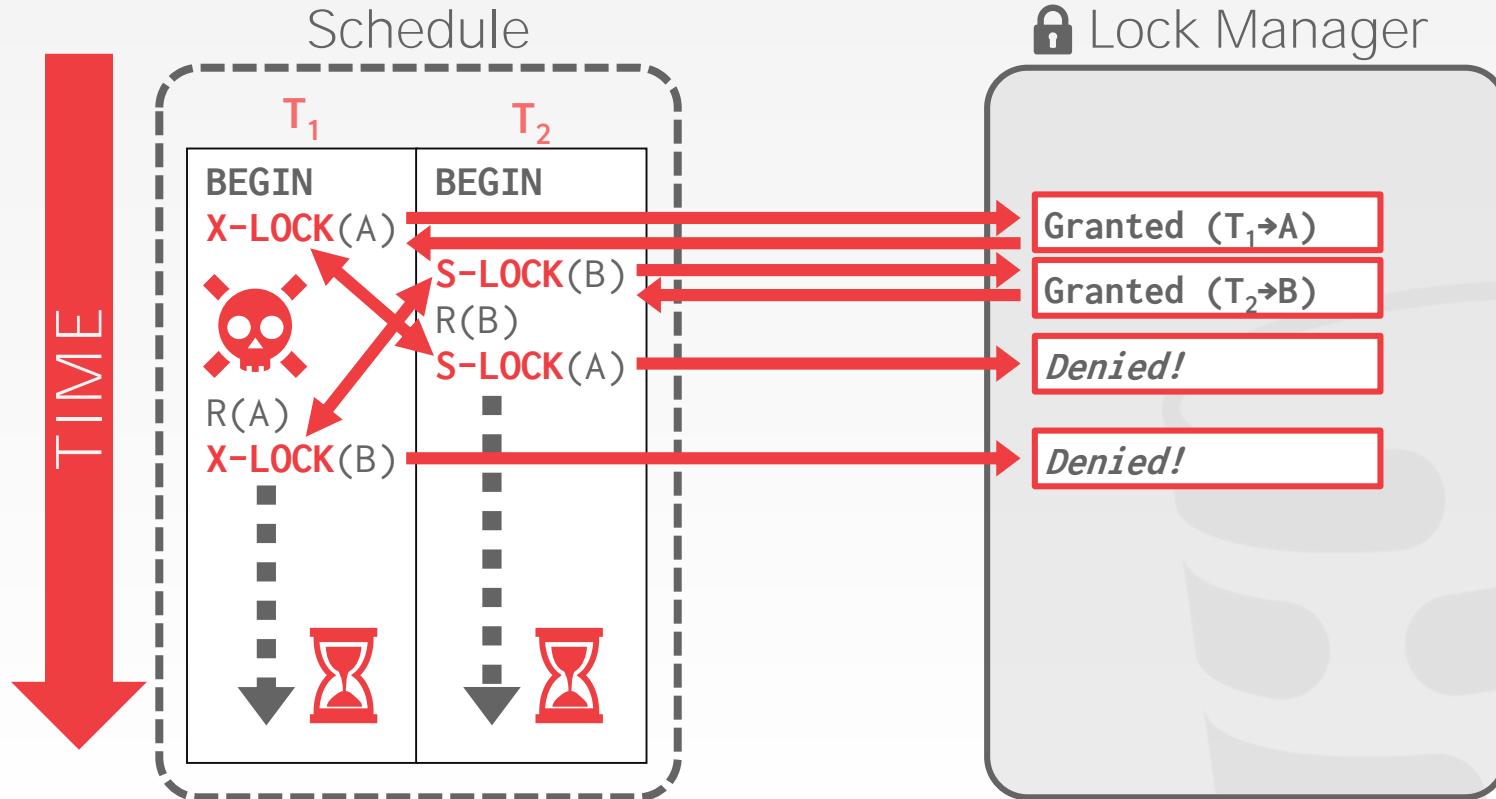
SHIT JUST GOT REAL, SON



SHIT JUST GOT REAL, SON



SHIT JUST GOT REAL, SON



2PL DEADLOCKS

A **deadlock** is a cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

- Approach #1: Deadlock Detection
- Approach #2: Deadlock Prevention



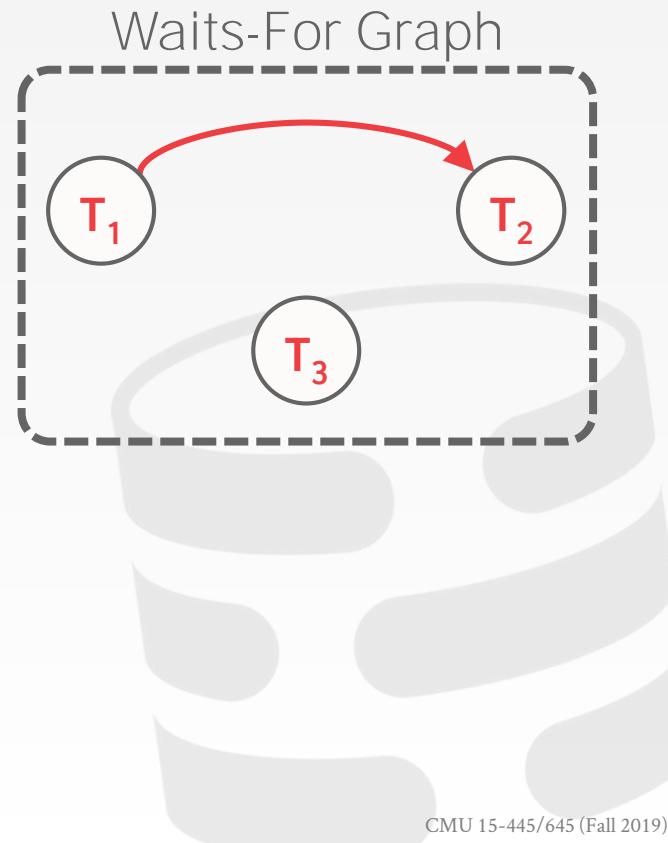
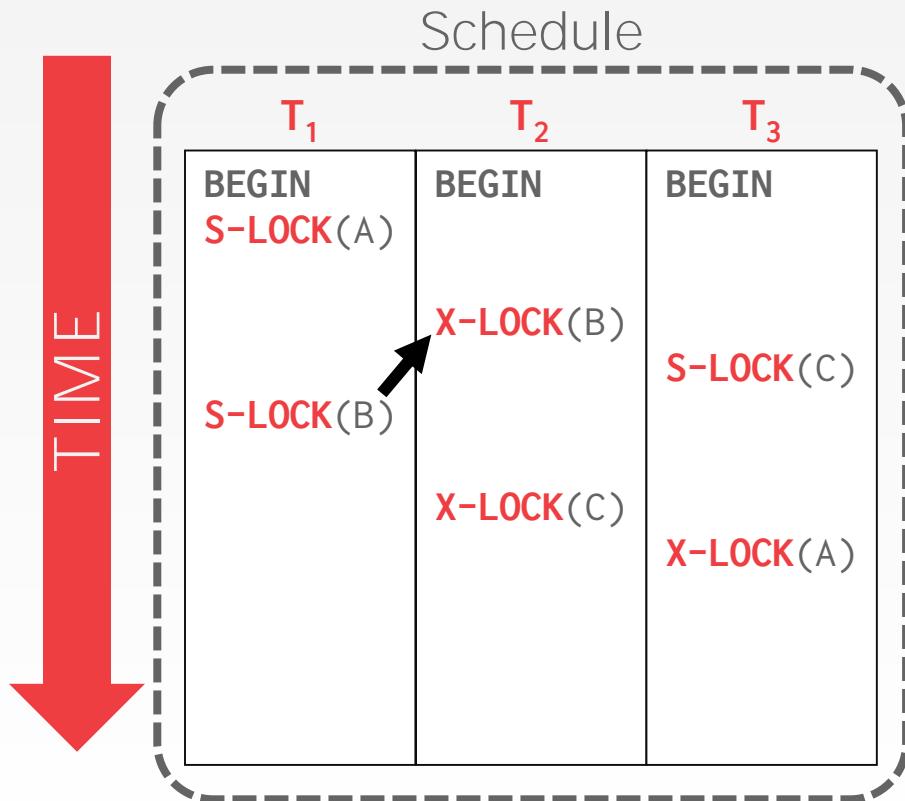
DEADLOCK DETECTION

The DBMS creates a waits-for graph to keep track of what locks each txn is waiting to acquire:

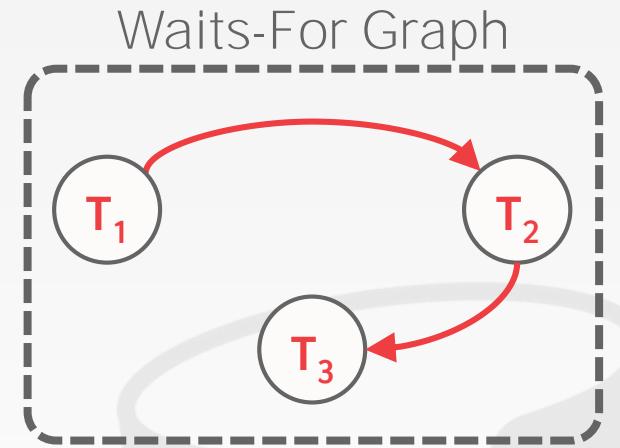
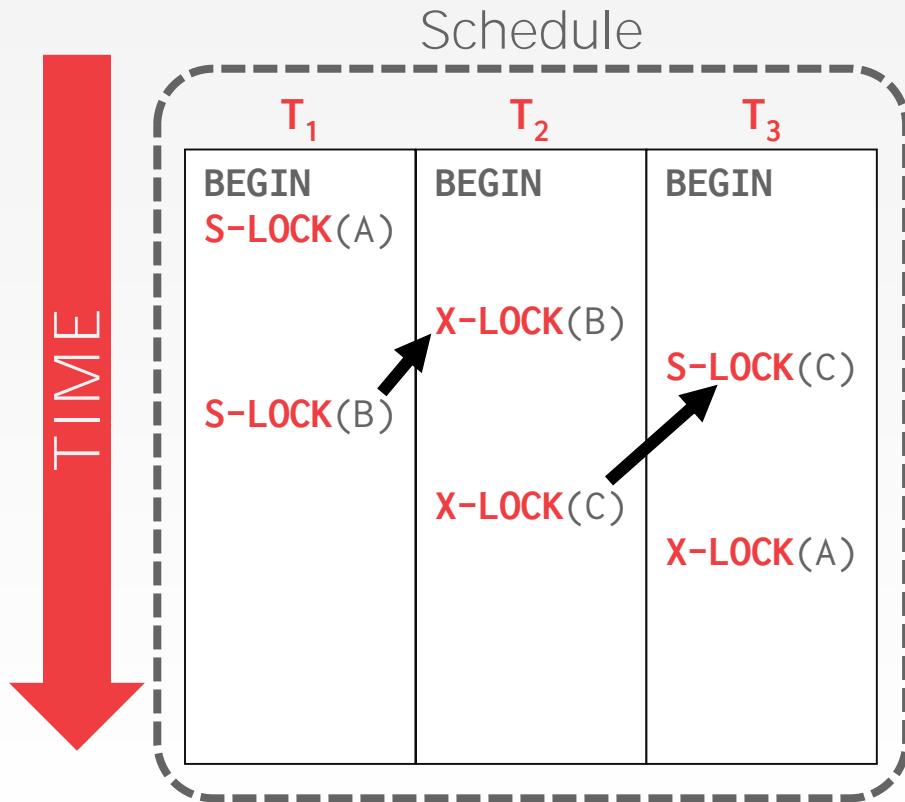
- Nodes are transactions
- Edge from T_i to T_j if T_i is waiting for T_j to release a lock.

The system periodically checks for cycles in *waits-for* graph and then decides how to break it.

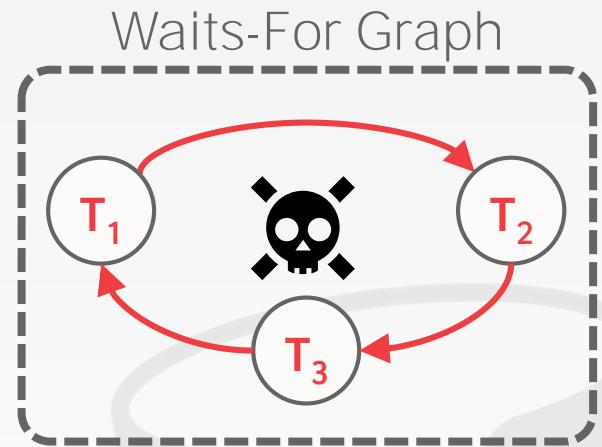
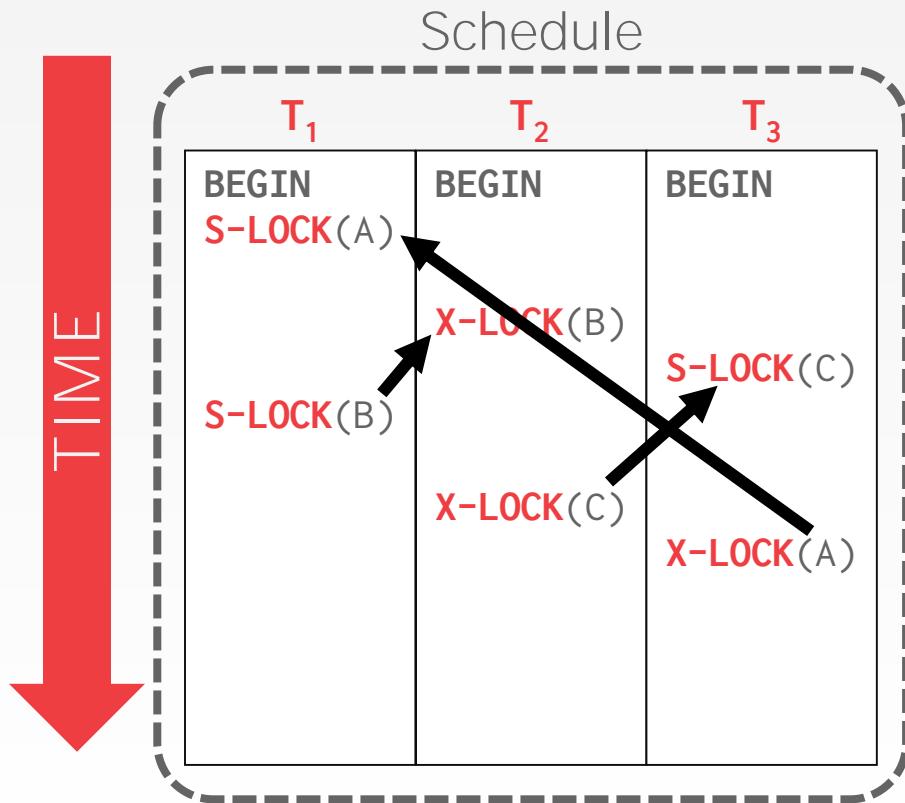
DEADLOCK DETECTION



DEADLOCK DETECTION



DEADLOCK DETECTION



DEADLOCK HANDLING

When the DBMS detects a deadlock, it will select a "victim" txn to rollback to break the cycle.

The victim txn will either restart or abort (more common) depending on how it was invoked.

There is a trade-off between the frequency of checking for deadlocks and how long txns have to wait before deadlocks are broken.

DEADLOCK HANDLING: VICTIM SELECTION

Selecting the proper victim depends on a lot of different variables....

- By age (lowest timestamp)
- By progress (least/most queries executed)
- By the # of items already locked
- By the # of txns that we have to rollback with it

We also should consider the # of times a txn has been restarted in the past to prevent starvation.

DEADLOCK HANDLING: ROLLBACK LENGTH

After selecting a victim txn to abort, the DBMS can also decide on how far to rollback the txn's changes.

Approach #1: Completely

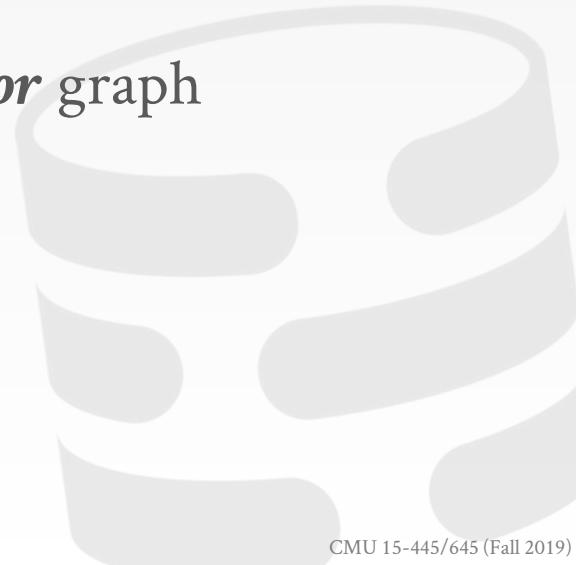
Approach #2: Minimally



DEADLOCK PREVENTION

When a txn tries to acquire a lock that is held by another txn, the DBMS kills one of them to prevent a deadlock.

This approach does not require a *waits-for* graph or detection algorithm.



DEADLOCK PREVENTION

Assign priorities based on timestamps:

- Older Timestamp = Higher Priority (e.g., $T_1 > T_2$)

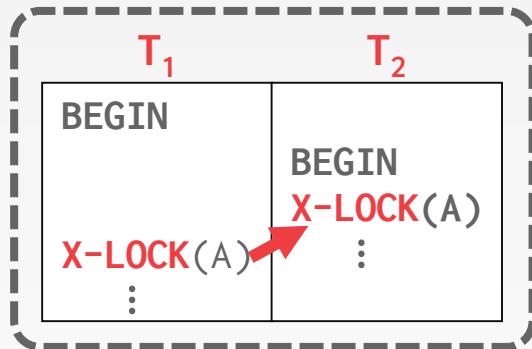
Wait-Die ("Old Waits for Young")

- If *requesting txn* has higher priority than *holding txn*, then *requesting txn* waits for *holding txn*.
- Otherwise *requesting txn* aborts.

Wound-Wait ("Young Waits for Old")

- If *requesting txn* has higher priority than *holding txn*, then *holding txn* aborts and releases lock.
- Otherwise *requesting txn* waits.

DEADLOCK PREVENTION

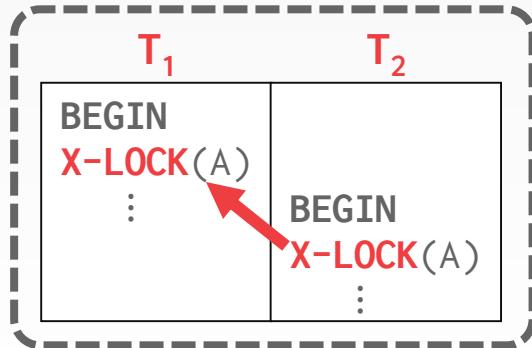


Wait-Die

T_1 waits

Wound-Wait

T_2 aborts



Wait-Die

T_2 aborts

Wound-Wait

T_2 waits

DEADLOCK PREVENTION

Why do these schemes guarantee no deadlocks?

Only one "type" of direction allowed when waiting for a lock.

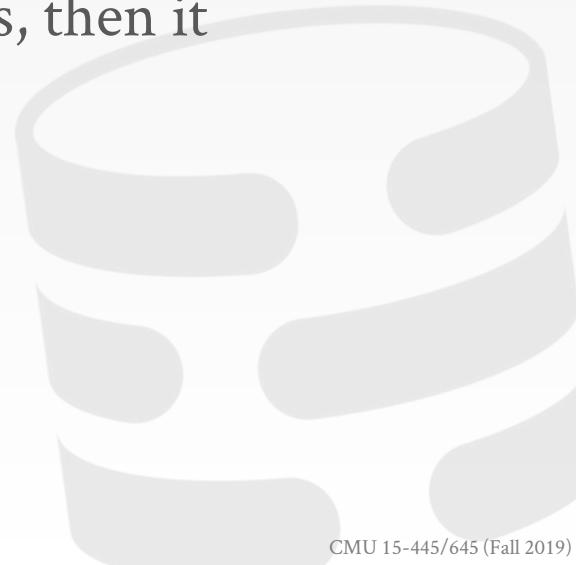
When a txn restarts, what is its (new) priority?

Its original timestamp. Why?

OBSERVATION

All of these examples have a one-to-one mapping from database objects to locks.

If a txn wants to update one billion tuples, then it has to acquire one billion locks.

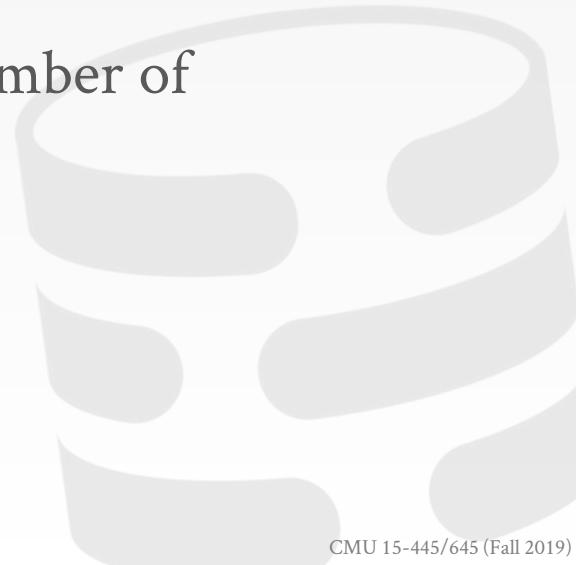


LOCK GRANULARITIES

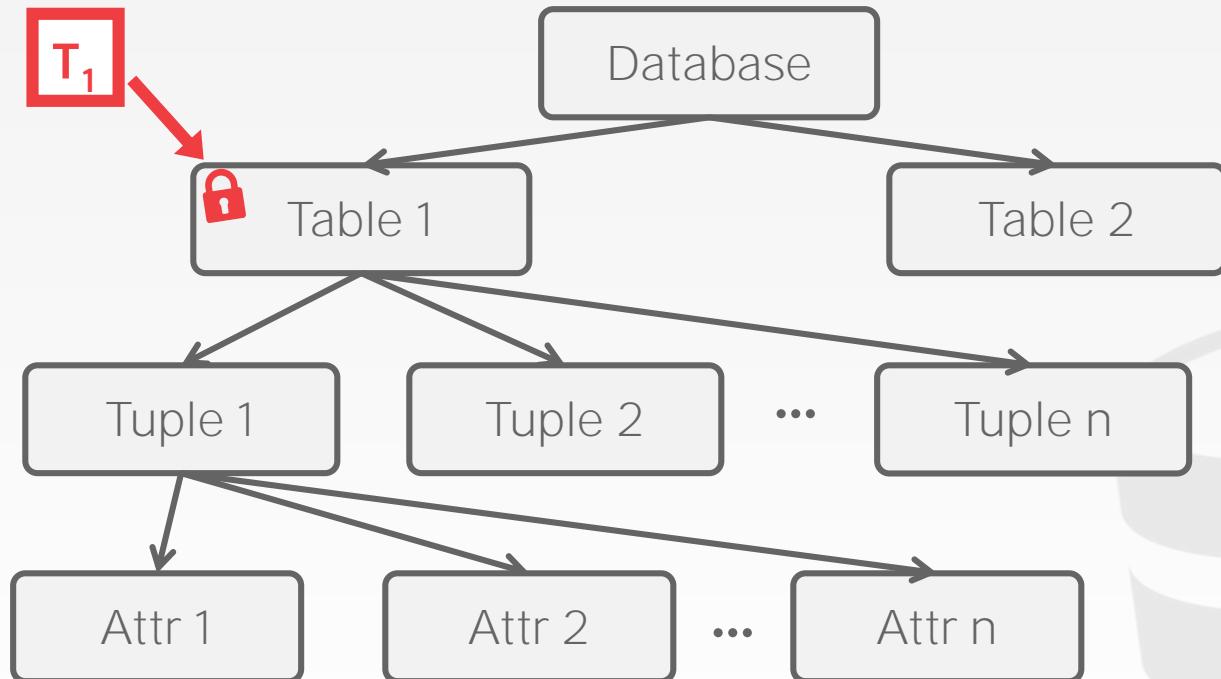
When we say that a txn acquires a “lock”, what does that actually mean?

→ On an Attribute? Tuple? Page? Table?

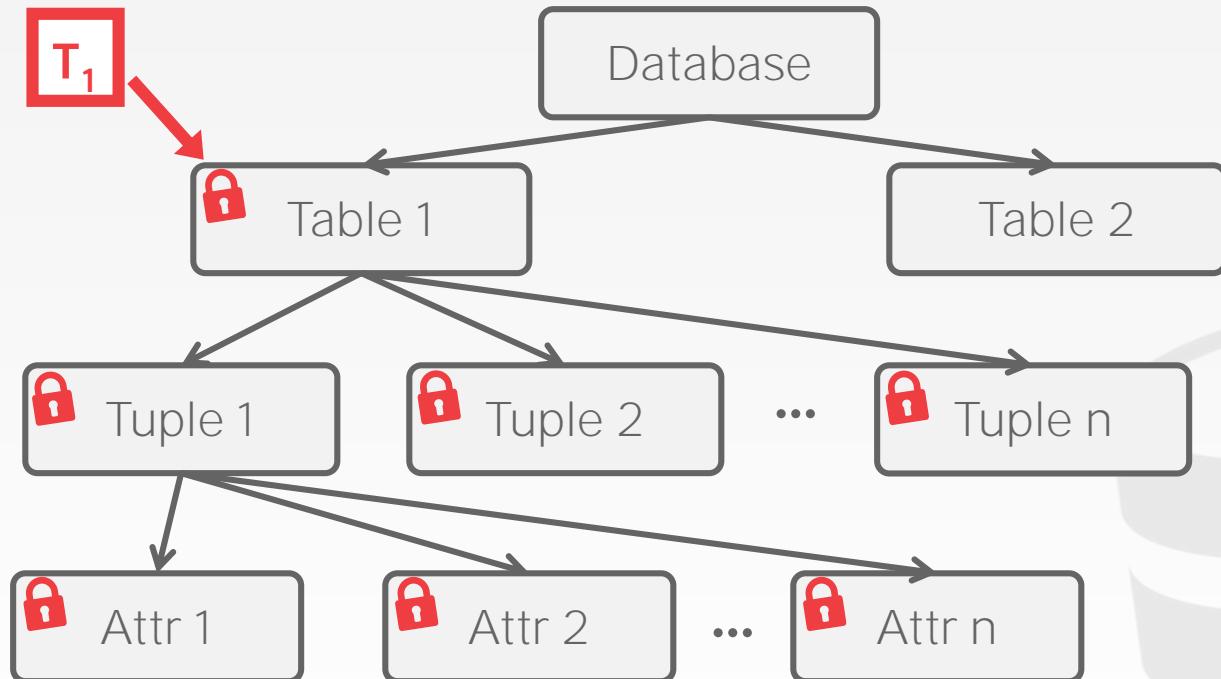
Ideally, each txn should obtain fewest number of locks that is needed...



DATABASE LOCK HIERARCHY



DATABASE LOCK HIERARCHY



EXAMPLE

T₁ – Get the balance of Andy's shady off-shore bank account.

T₂ – Increase Matt's bank account balance by 1%.

What locks should these txns obtain?



EXAMPLE

T₁ – Get the balance of Andy's shady off-shore bank account.

T₂ – Increase Matt's bank account balance by 1%.

What locks should these txns obtain?

Multiple:

- Exclusive + Shared for leafs of lock tree.
- Special Intention locks for higher levels.



INTENTION LOCKS

An intention lock allows a higher level node to be locked in **shared** or **exclusive** mode without having to check all descendent nodes.

If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.

INTENTION LOCKS

Intention-Shared (IS)

- Indicates explicit locking at a lower level with shared locks.

Intention-Exclusive (IX)

- Indicates locking at lower level with exclusive or shared locks.



INTENTION LOCKS

Shared+Intention-Exclusive (**SIX**)

- The subtree rooted by that node is locked explicitly in **shared** mode and explicit locking is being done at a lower level with **exclusive-mode** locks.



COMPATIBILITY MATRIX

		T_2 Wants				
		IS	IX	S	SIX	X
T_1 Holds	IS	✓	✓	✓	✓	✗
	IX	✓	✓	✗	✗	✗
	S	✓	✗	✓	✗	✗
	SIX	✓	✗	✗	✗	✗
	X	✗	✗	✗	✗	✗

LOCKING PROTOCOL

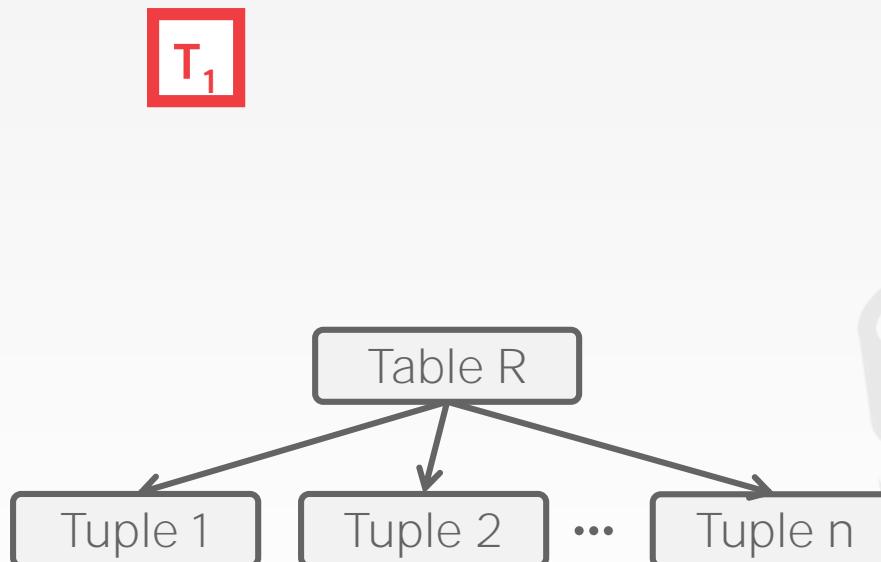
Each txn obtains appropriate lock at highest level of the database hierarchy.

To get **S** or **IS** lock on a node, the txn must hold at least **IS** on parent node.

To get **X**, **IX**, or **SIX** on a node, must hold at least **IX** on parent node.

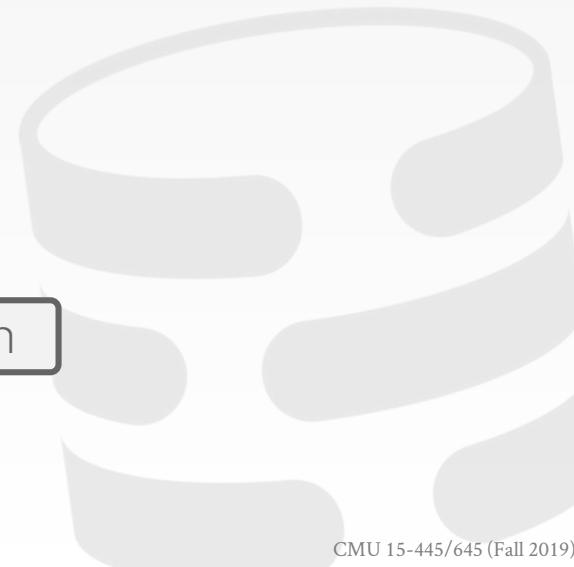
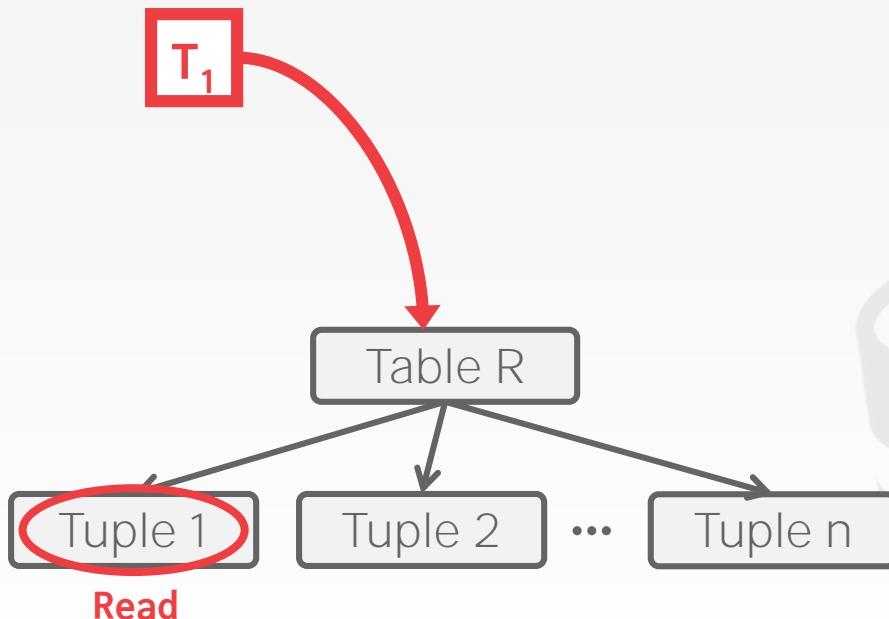
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.



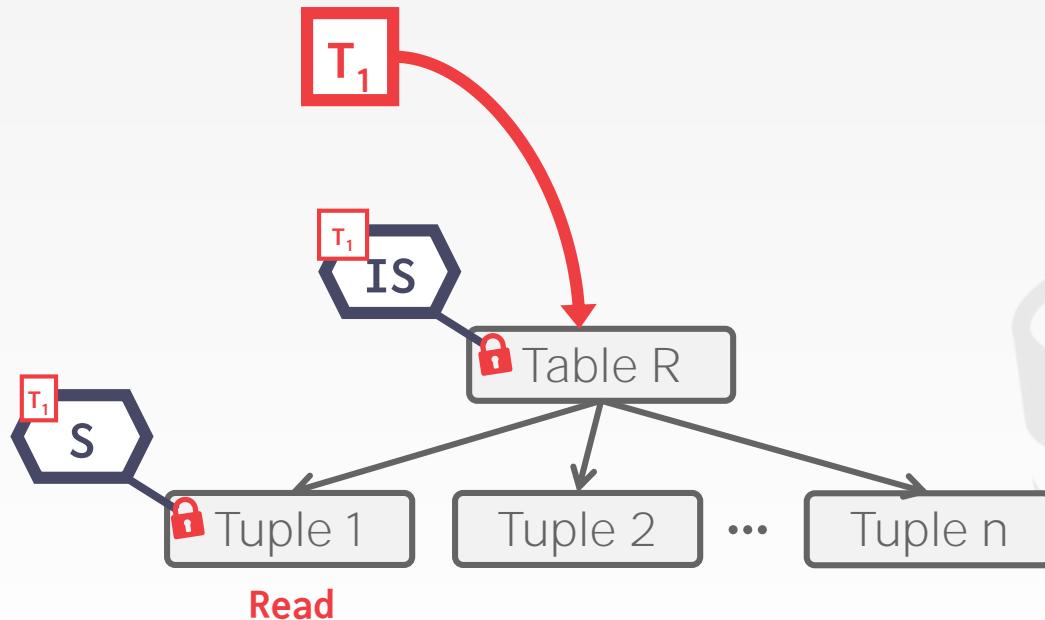
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.



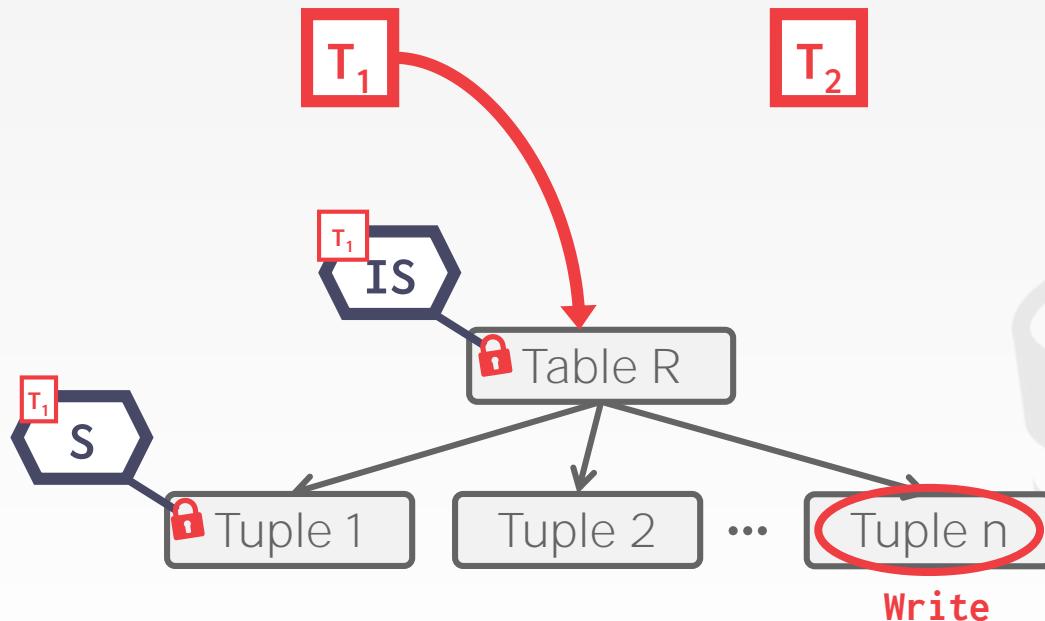
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.



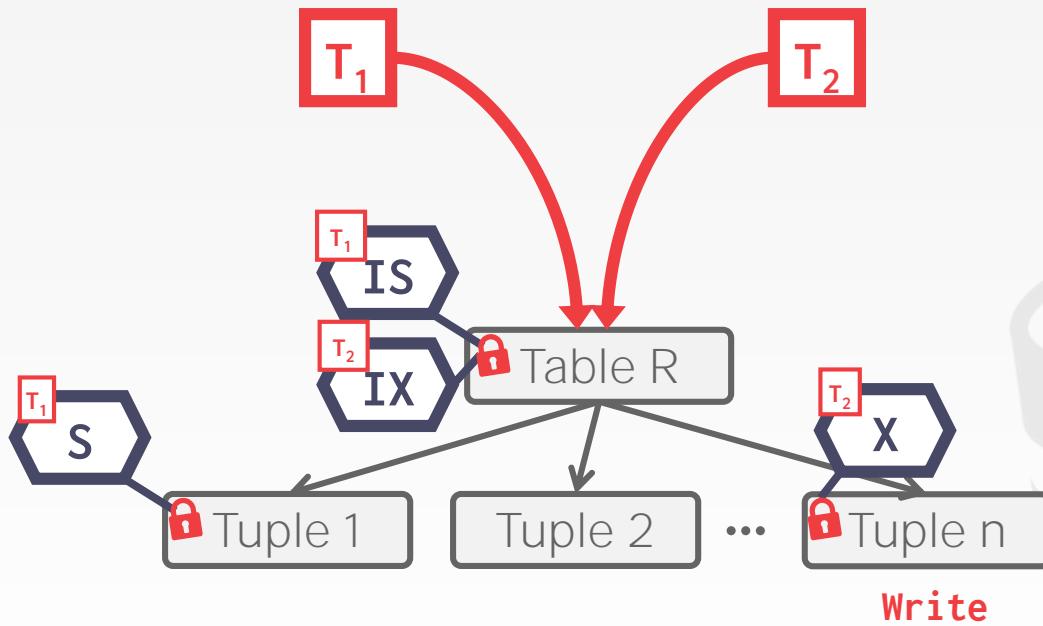
EXAMPLE – TWO-LEVEL HIERARCHY

Update Matt's record in R.



EXAMPLE – TWO-LEVEL HIERARCHY

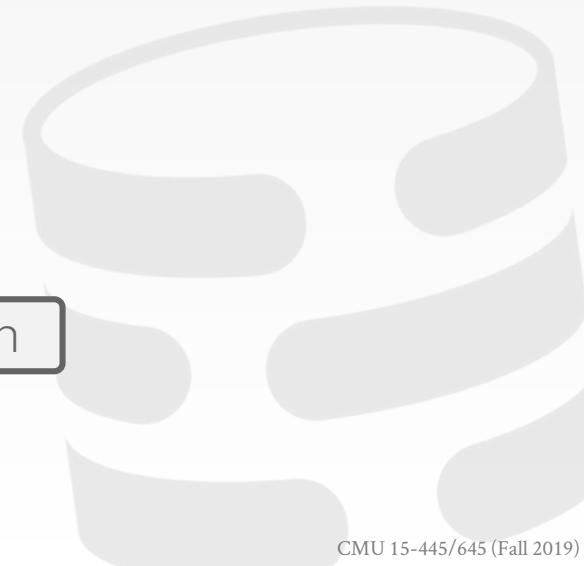
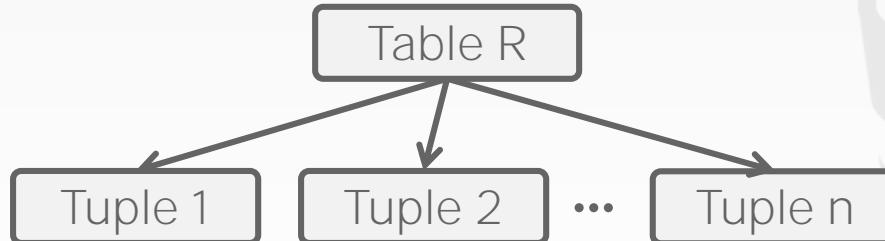
Update Matt's record in R.



EXAMPLE – THREESOME

Assume three txns execute at same time:

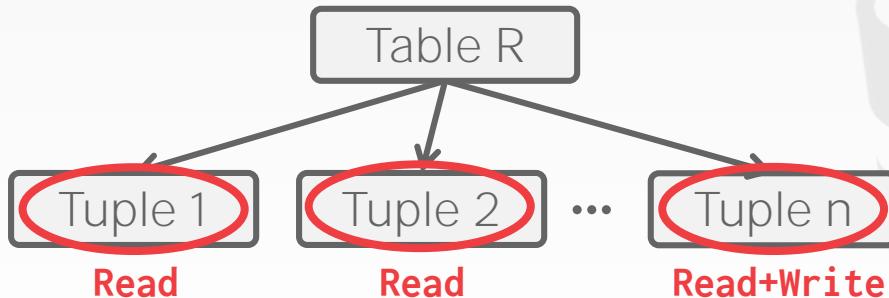
- T_1 – Scan **R** and update a few tuples.
- T_2 – Read a single tuple in **R**.
- T_3 – Scan all tuples in **R**.



EXAMPLE – THREESOME

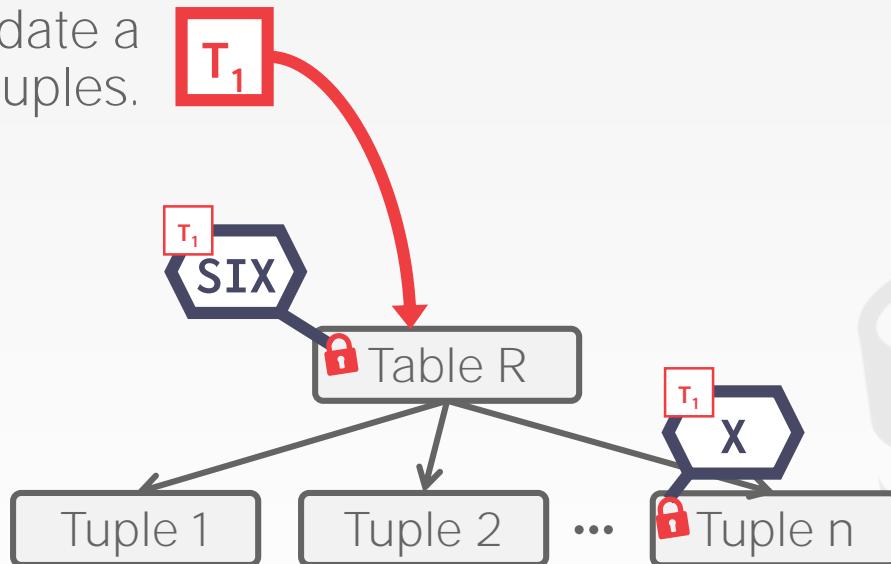
Scan **R** and update a few tuples.

T₁

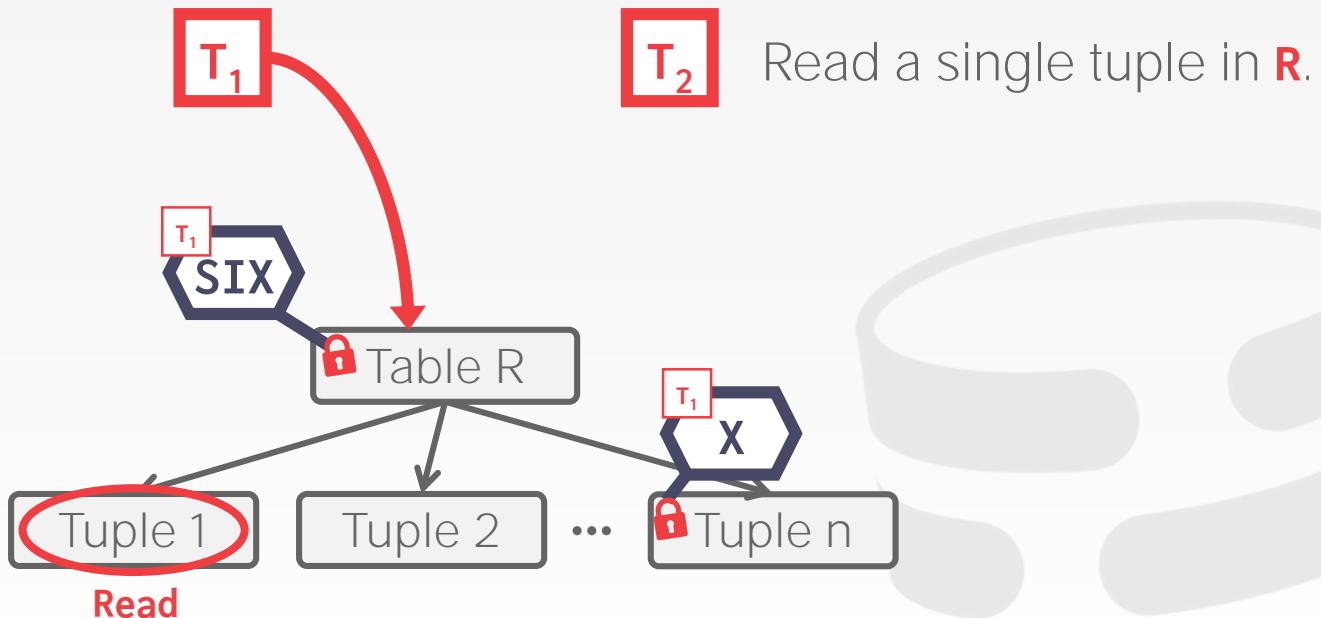


EXAMPLE – THREESOME

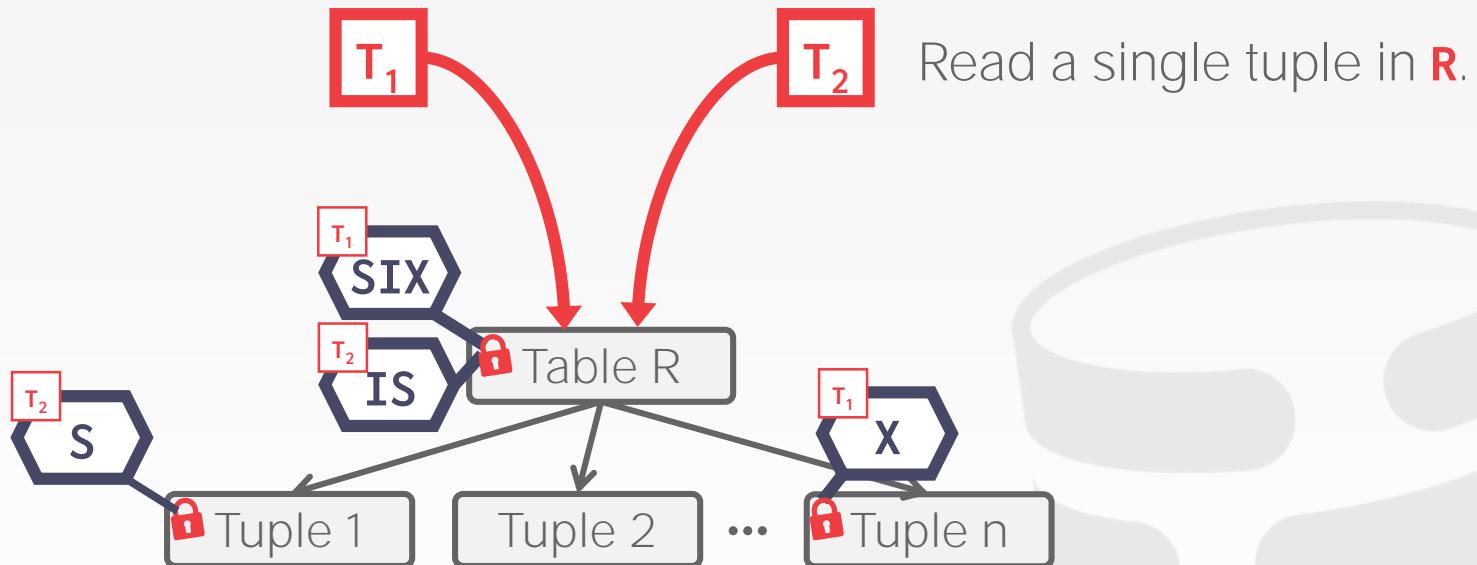
Scan **R** and update a few tuples.



EXAMPLE – THREESOME

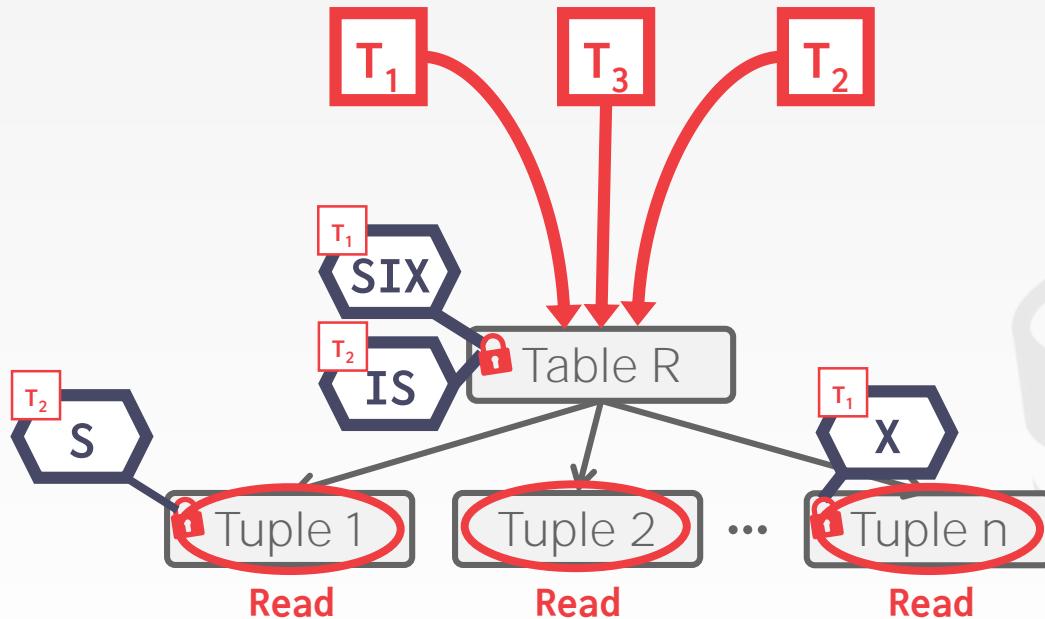


EXAMPLE – THREESOME



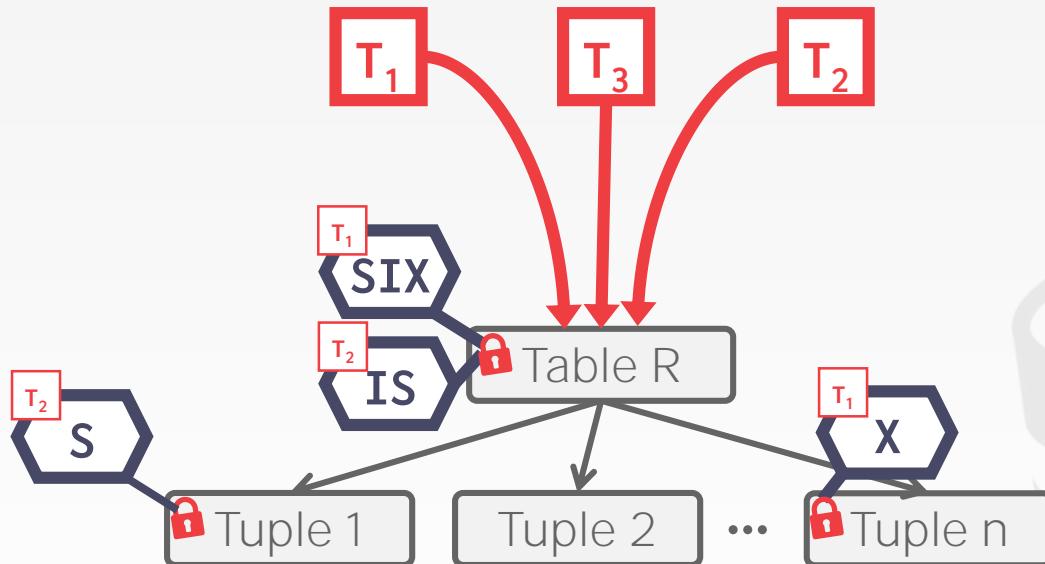
EXAMPLE – THREESOME

Scan all tuples in **R**.



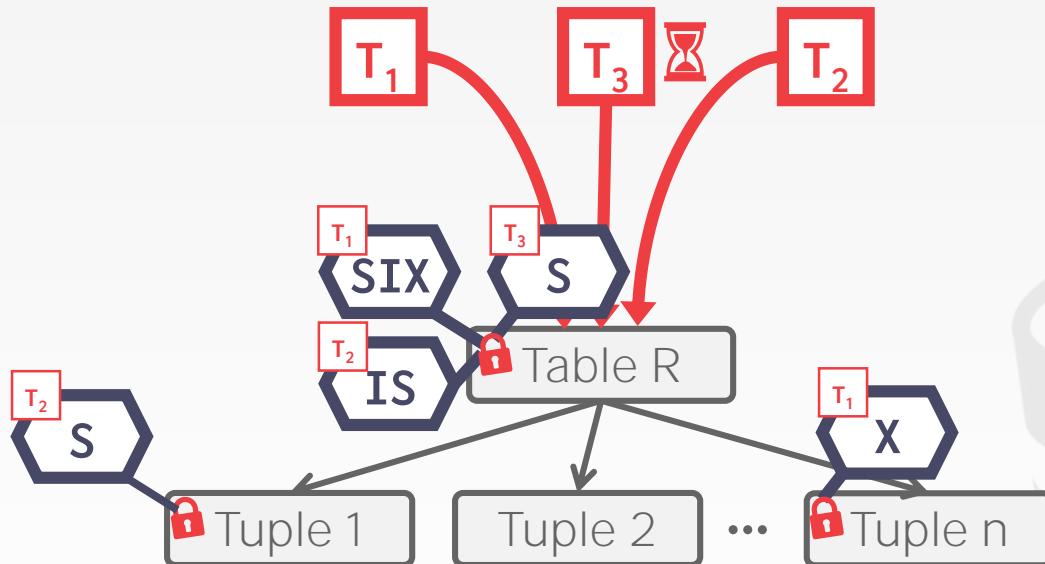
EXAMPLE – THREESOME

Scan all tuples in **R**.



EXAMPLE – THREESOME

Scan all tuples in **R**.



MULTIPLE LOCK GRANULARITIES

Hierarchical locks are useful in practice as each txn only needs a few locks.

Intention locks help improve concurrency:

- **Intention-Shared (IS)**: Intent to get **S** lock(s) at finer granularity.
- **Intention-Exclusive (IX)**: Intent to get **X** lock(s) at finer granularity.
- **Shared+Intention-Exclusive (SIX)**: Like **S** and **IX** at the same time.

LOCK ESCALATION

Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired.

This reduces the number of requests that the lock manager has to process.



LOCKING IN PRACTICE

You typically don't set locks manually in txns.

Sometimes you will need to provide the DBMS with hints to help it to improve concurrency.

Explicit locks are also useful when doing major changes to the database.

LOCK TABLE

Explicitly locks a table.

Not part of the SQL standard.

- Postgres/DB2/Oracle Modes: **SHARE, EXCLUSIVE**
- MySQL Modes: **READ, WRITE**



```
LOCK TABLE <table> IN <mode> MODE;
```

```
SELECT 1 FROM <table> WITH (TABLOCK, <mode>);
```

```
LOCK TABLE <table> <mode>;
```

SELECT...FOR UPDATE

Perform a select and then sets an exclusive lock on the matching tuples.

Can also set shared locks:

- Postgres: **FOR SHARE**
- MySQL: **LOCK IN SHARE MODE**

```
SELECT * FROM <table>
WHERE <qualification> FOR UPDATE;
```

CONCLUSION

2PL is used in almost DBMS.

Automatically generates correct interleaving:

- Locks + protocol (2PL, SS2PL ...)
- Deadlock detection + handling
- Deadlock prevention



NEXT CLASS

Timestamp Ordering Concurrency Control



18 |

Timestamp Ordering Concurrency Control



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

CONCURRENCY CONTROL APPROACHES

Two-Phase Locking (2PL)

- Determine serializability order of conflicting operations at runtime while txns execute.

Timestamp Ordering (T/O)

- Determine serializability order of txns before they execute.

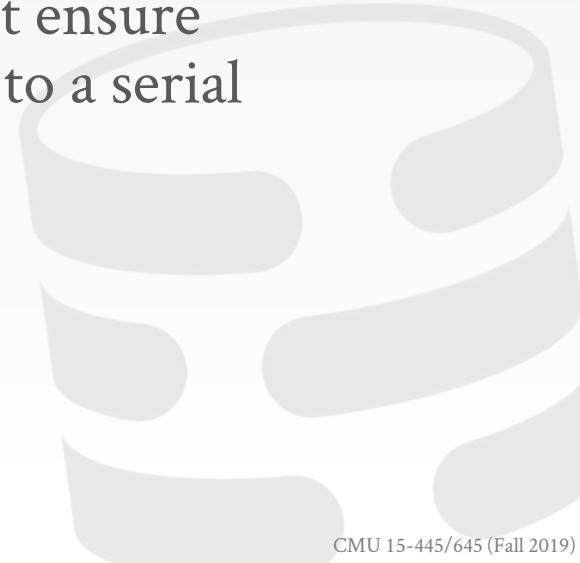
Pessimistic

Optimistic

T/O CONCURRENCY CONTROL

Use timestamps to determine the serializability order of txns.

If $TS(T_i) < TS(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where T_i appears before T_j .



TIMESTAMP ALLOCATION

Each txn T_i is assigned a unique fixed timestamp that is monotonically increasing.

- Let $TS(T_i)$ be the timestamp allocated to txn T_i .
- Different schemes assign timestamps at different times during the txn.

Multiple implementation strategies:

- System Clock.
- Logical Counter.
- Hybrid.



TODAY'S AGENDA

Basic Timestamp Ordering Protocol
Optimistic Concurrency Control
Partition-based Timestamp Ordering
Isolation Levels



BASIC T/O

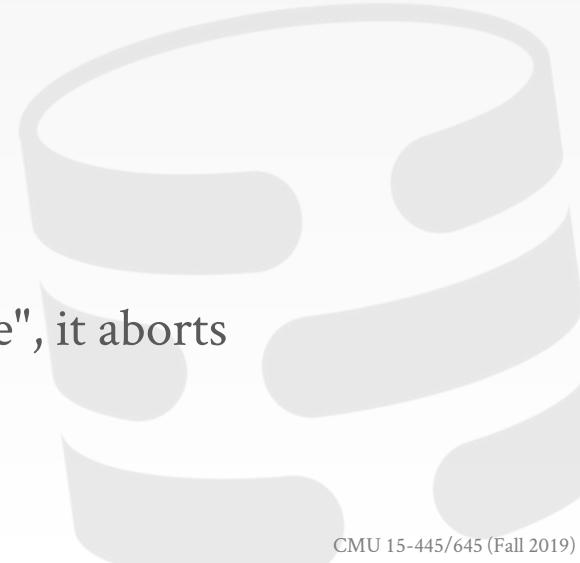
Txns read and write objects without locks.

Every object **X** is tagged with timestamp of the last txn that successfully did read/write:

- **W-TS(X)** – Write timestamp on **X**
- **R-TS(X)** – Read timestamp on **X**

Check timestamps for every operation:

- If txn tries to access an object "from the future", it aborts and restarts.



BASIC T/O – READS

If $TS(T_i) < W-TS(X)$, this violates timestamp order of T_i with regard to the writer of X .
→ Abort T_i and restart it with a newer TS.

Else:

- Allow T_i to read X .
- Update $R-TS(X)$ to $\max(R-TS(X), TS(T_i))$
- Have to make a local copy of X to ensure repeatable reads for T_i .



BASIC T/O - WRITES

If $TS(T_i) < R-TS(X)$ or $TS(T_i) < W-TS(X)$

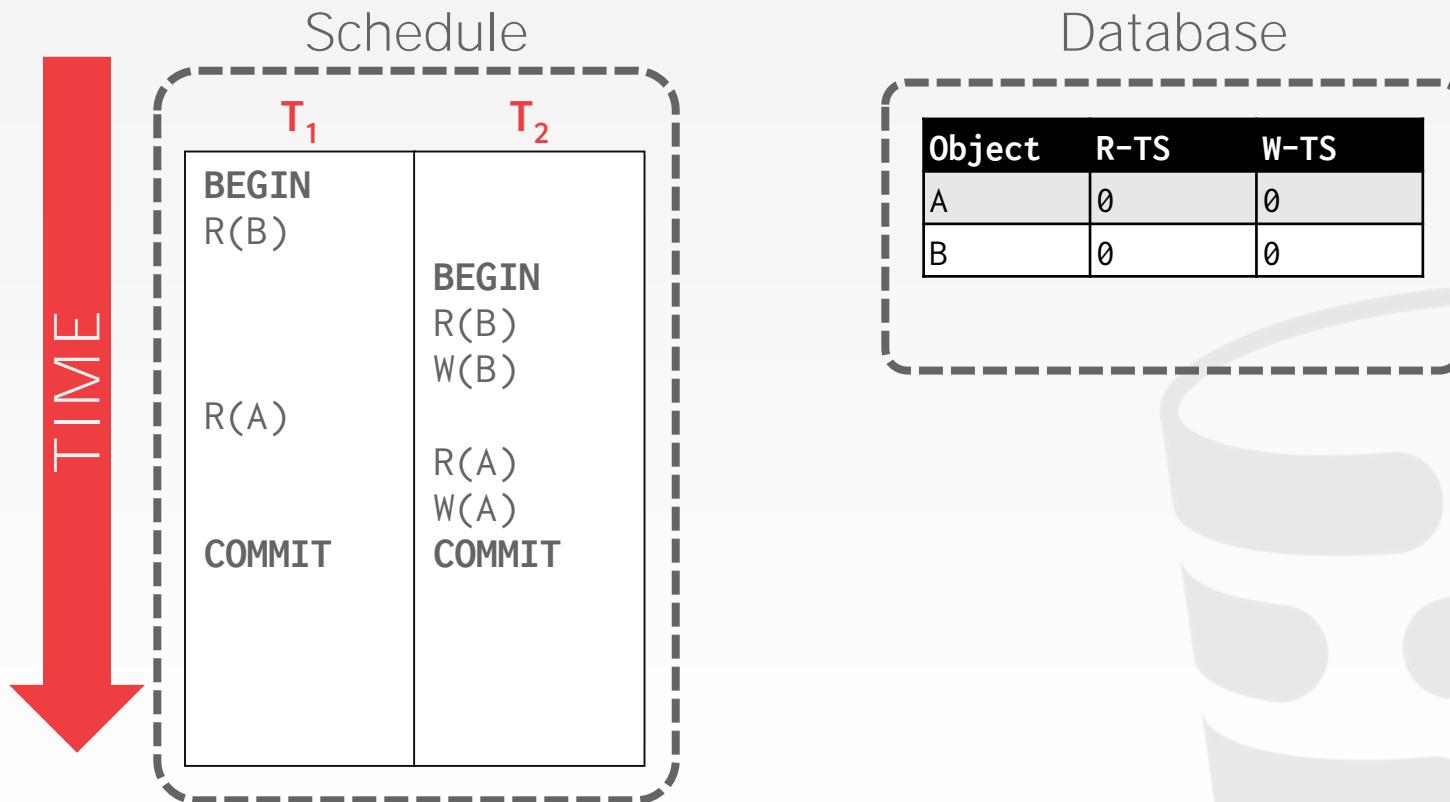
→ Abort and restart T_i .

Else:

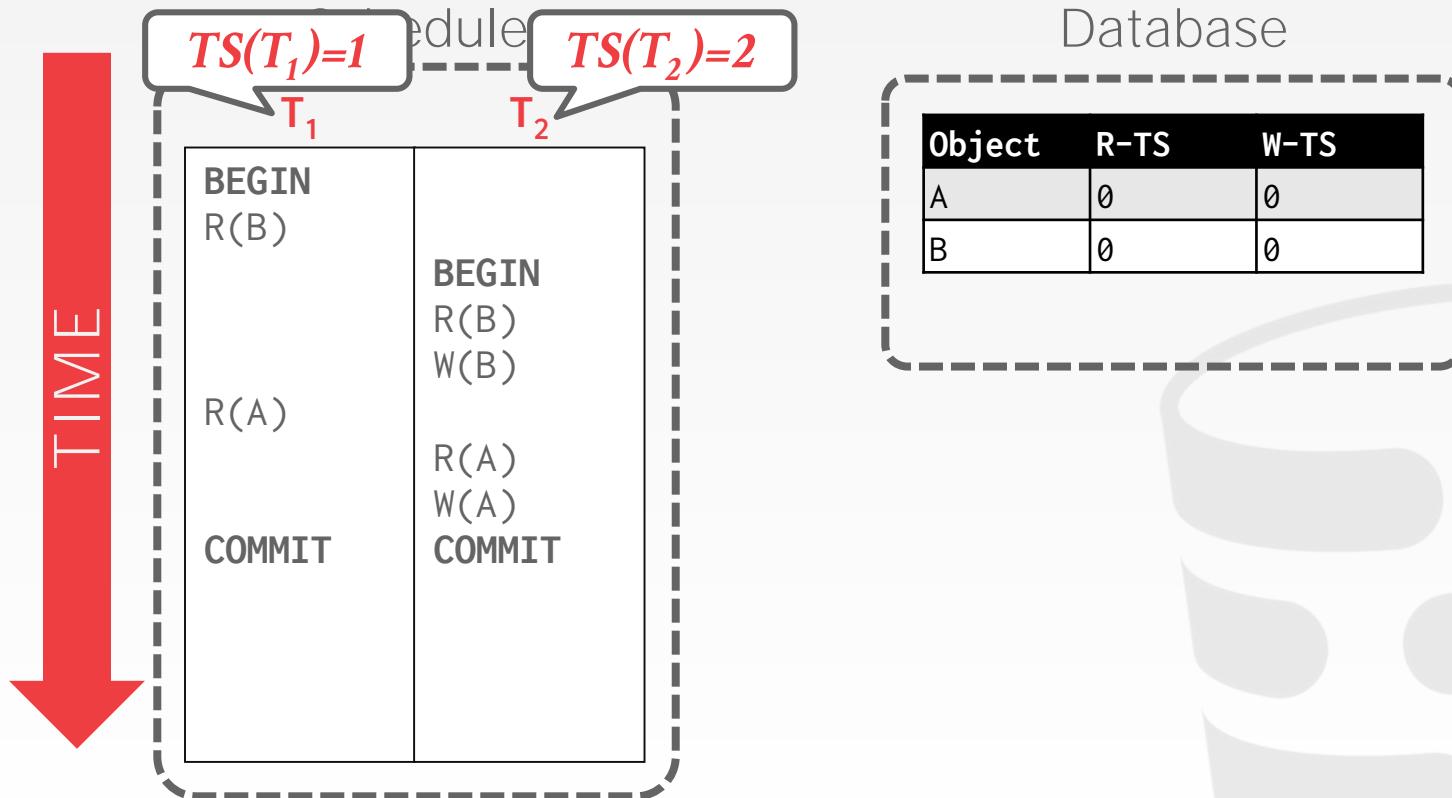
→ Allow T_i to write X and update $W-TS(X)$

→ Also have to make a local copy of X to ensure repeatable reads for T_i .

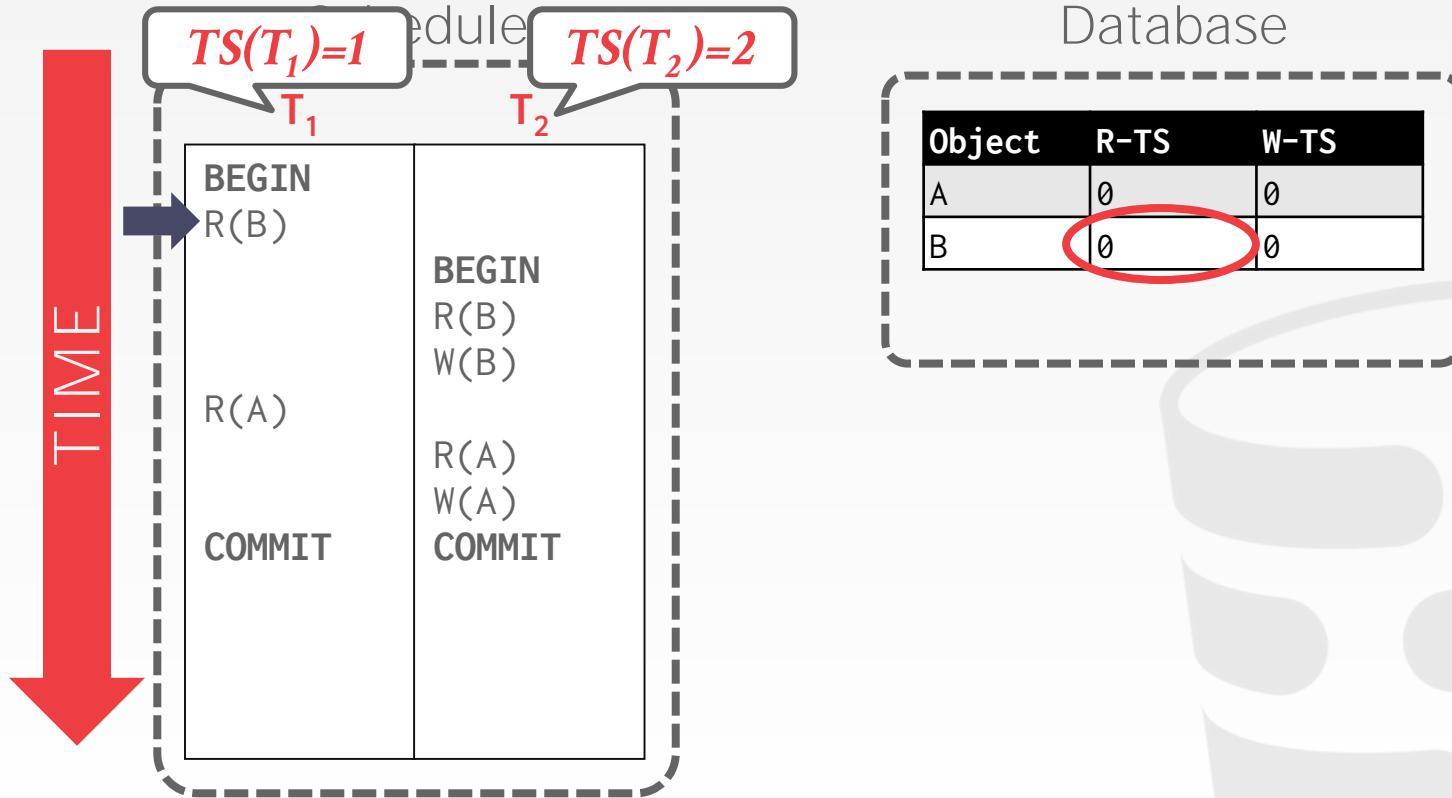
BASIC T/O - EXAMPLE #1



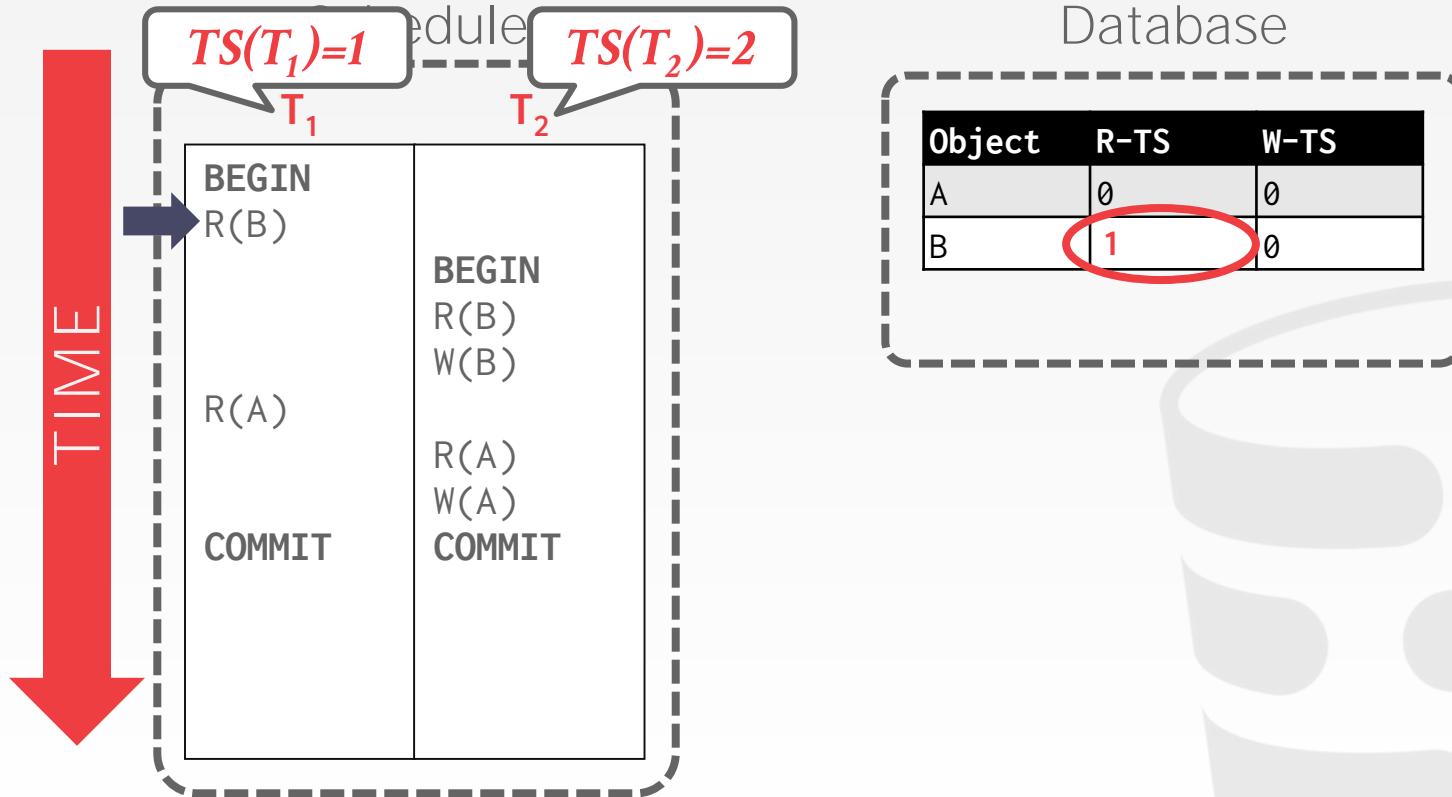
BASIC T/O - EXAMPLE #1



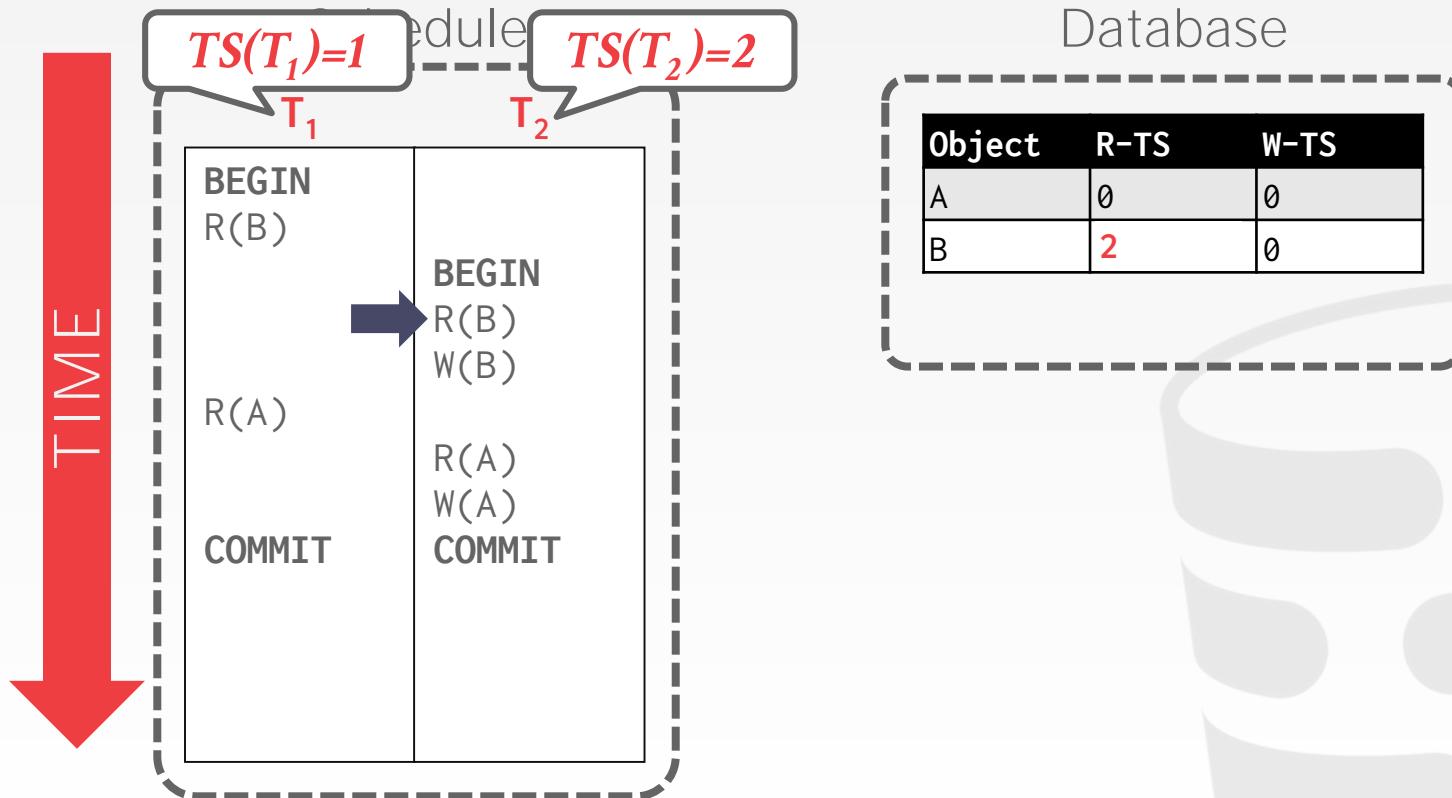
BASIC T/O - EXAMPLE #1



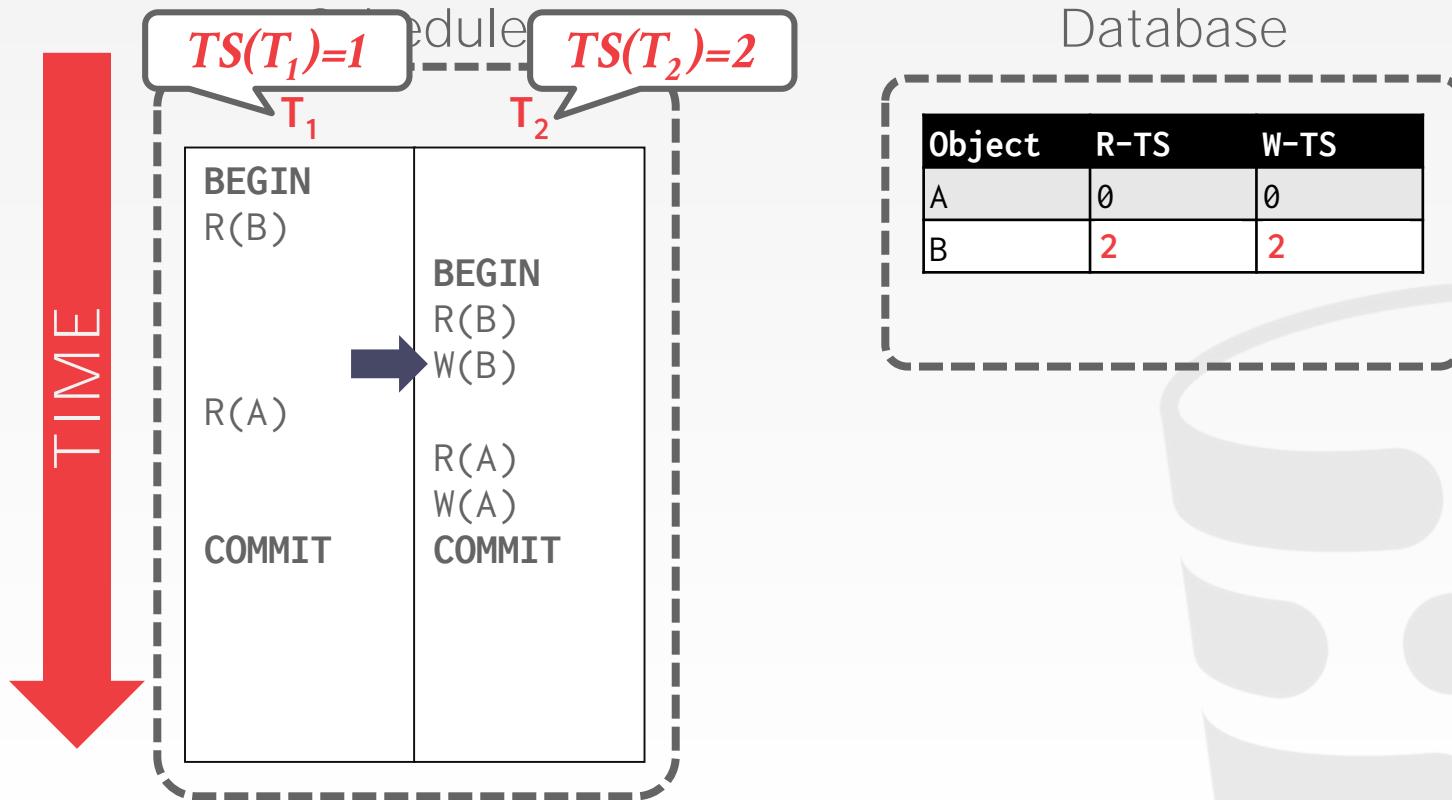
BASIC T/O - EXAMPLE #1



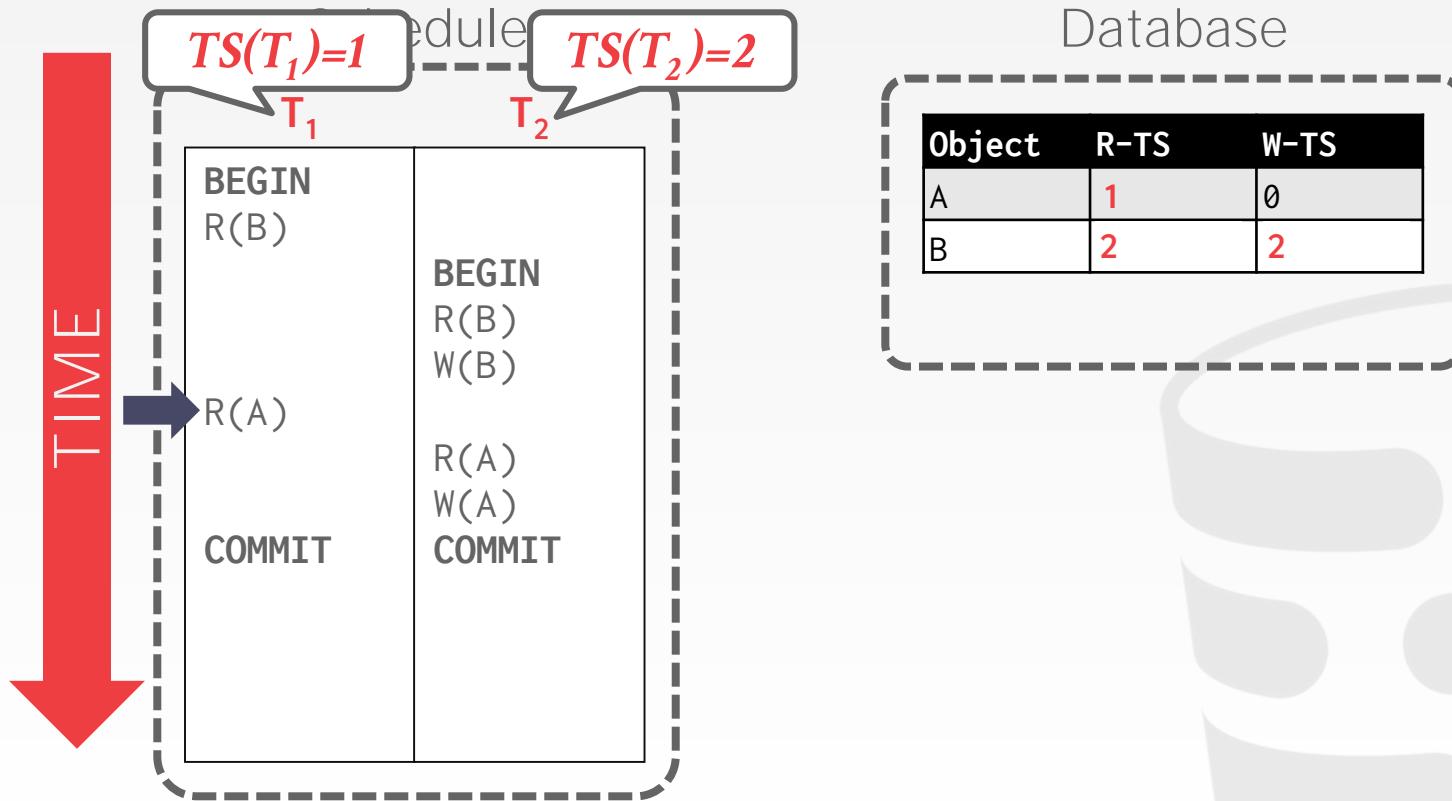
BASIC T/O - EXAMPLE #1



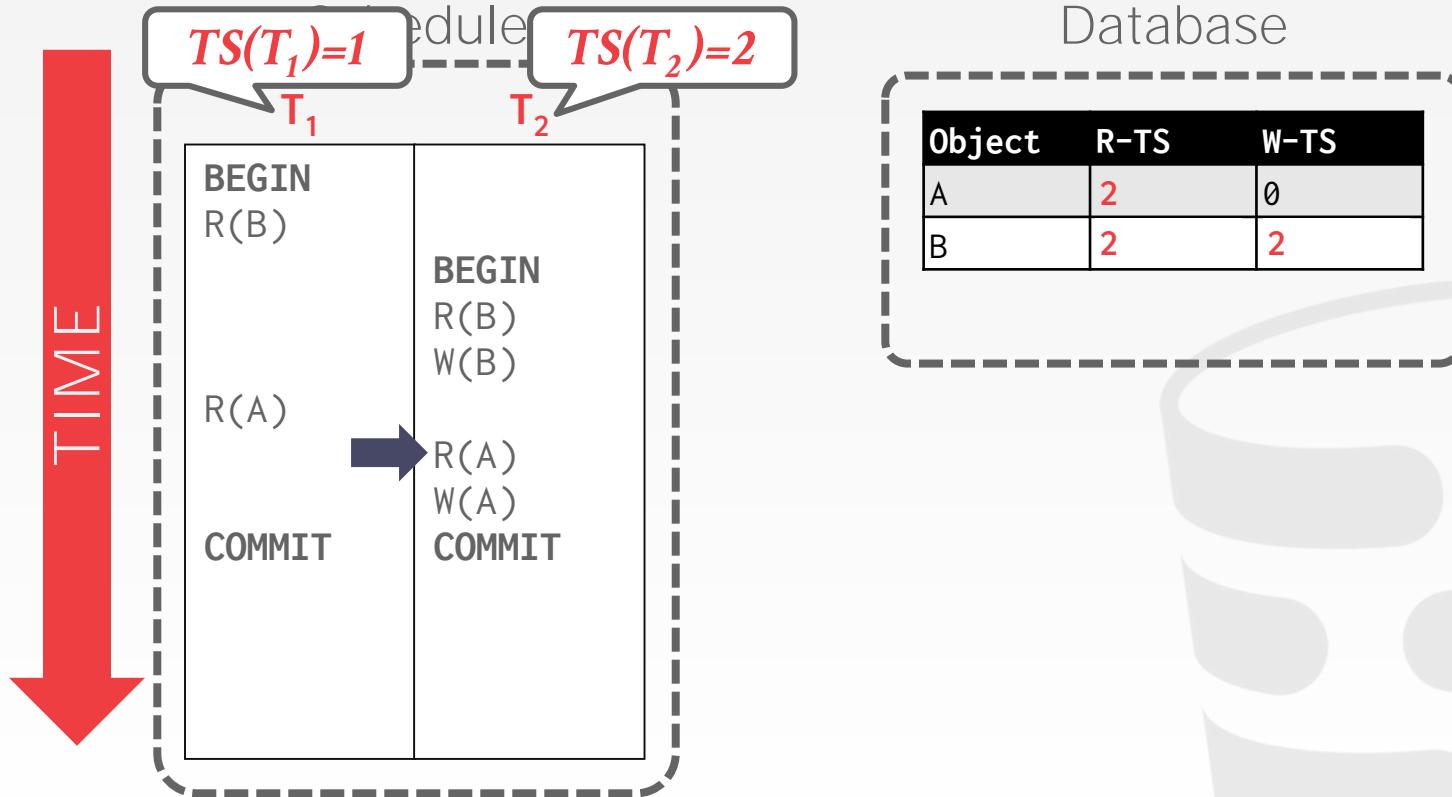
BASIC T/O - EXAMPLE #1



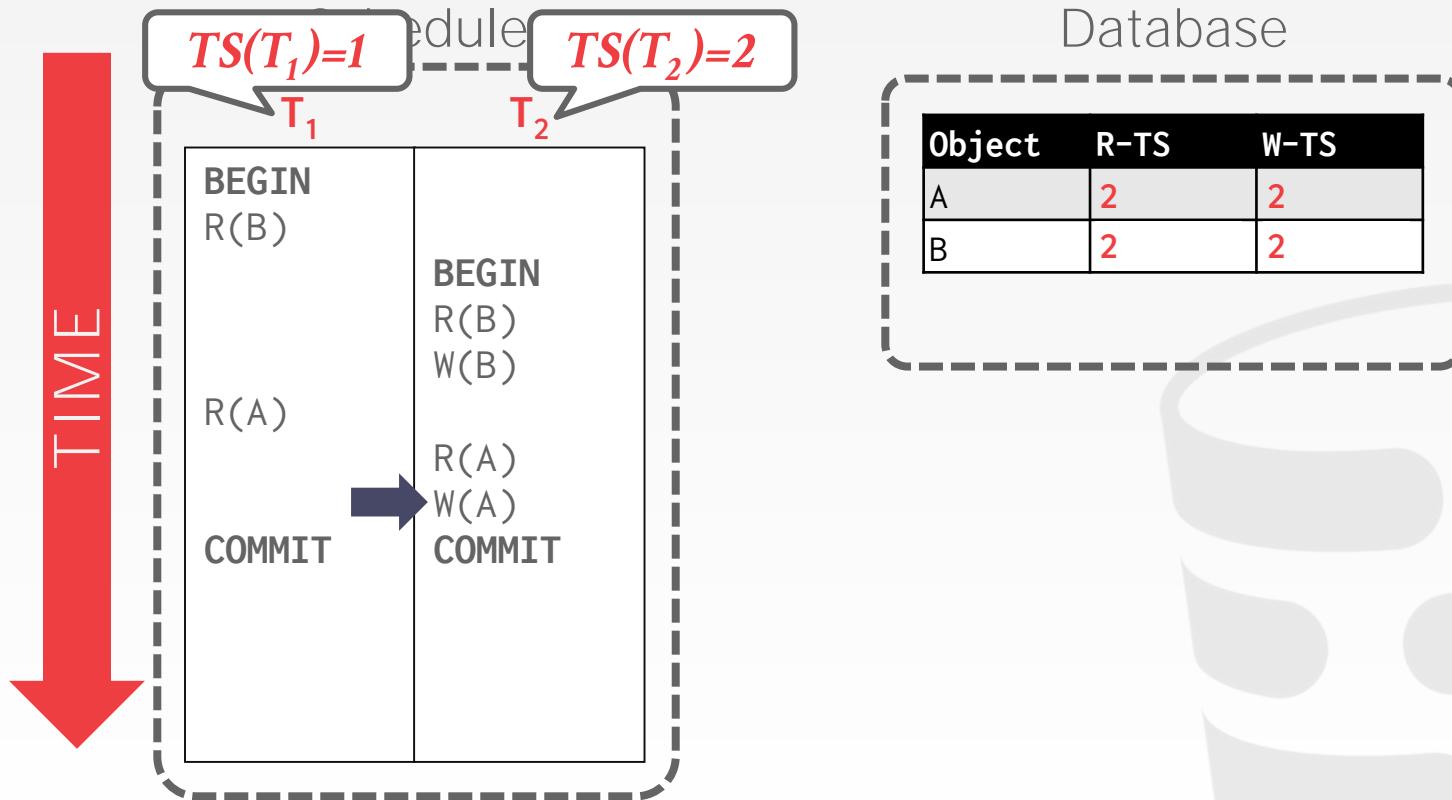
BASIC T/O - EXAMPLE #1



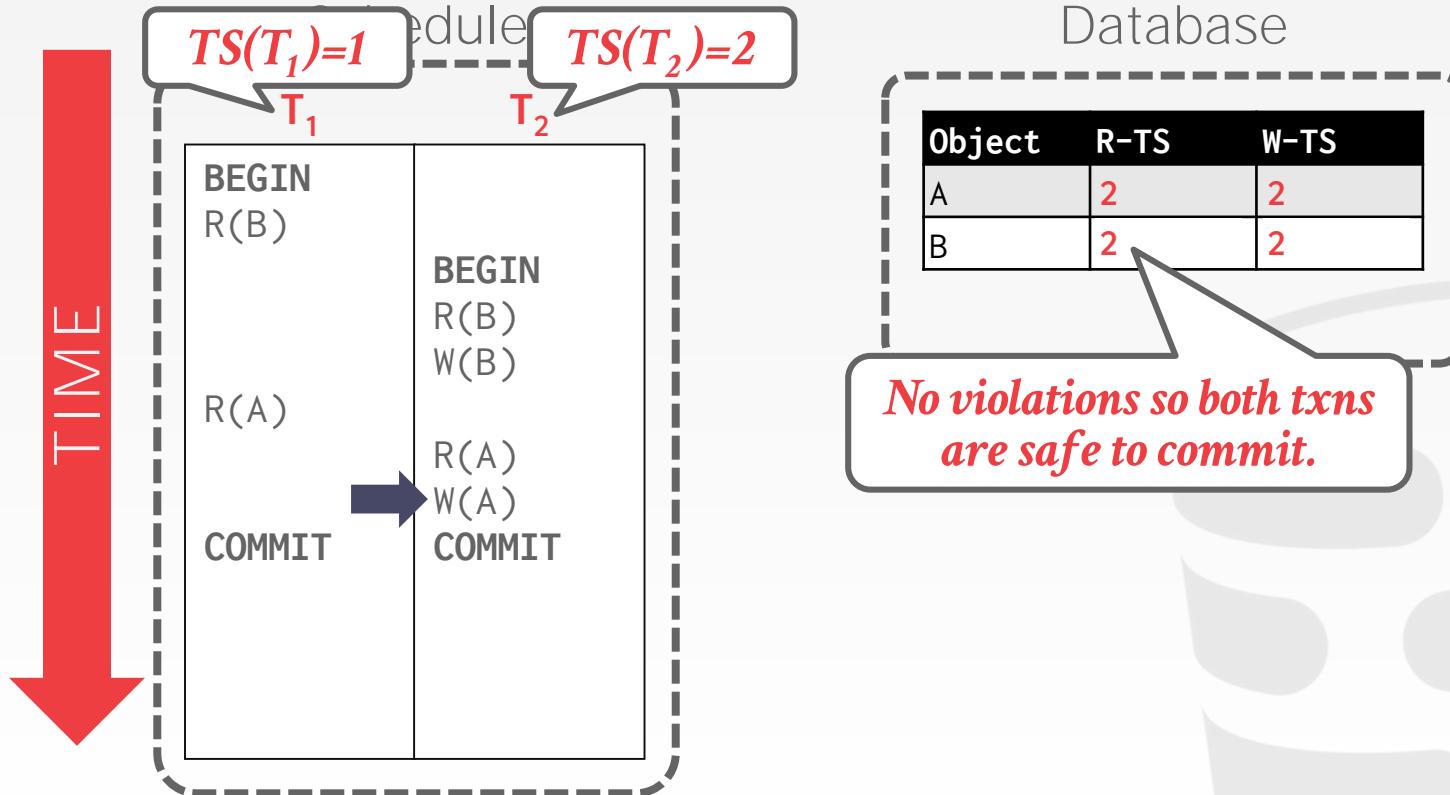
BASIC T/O - EXAMPLE #1



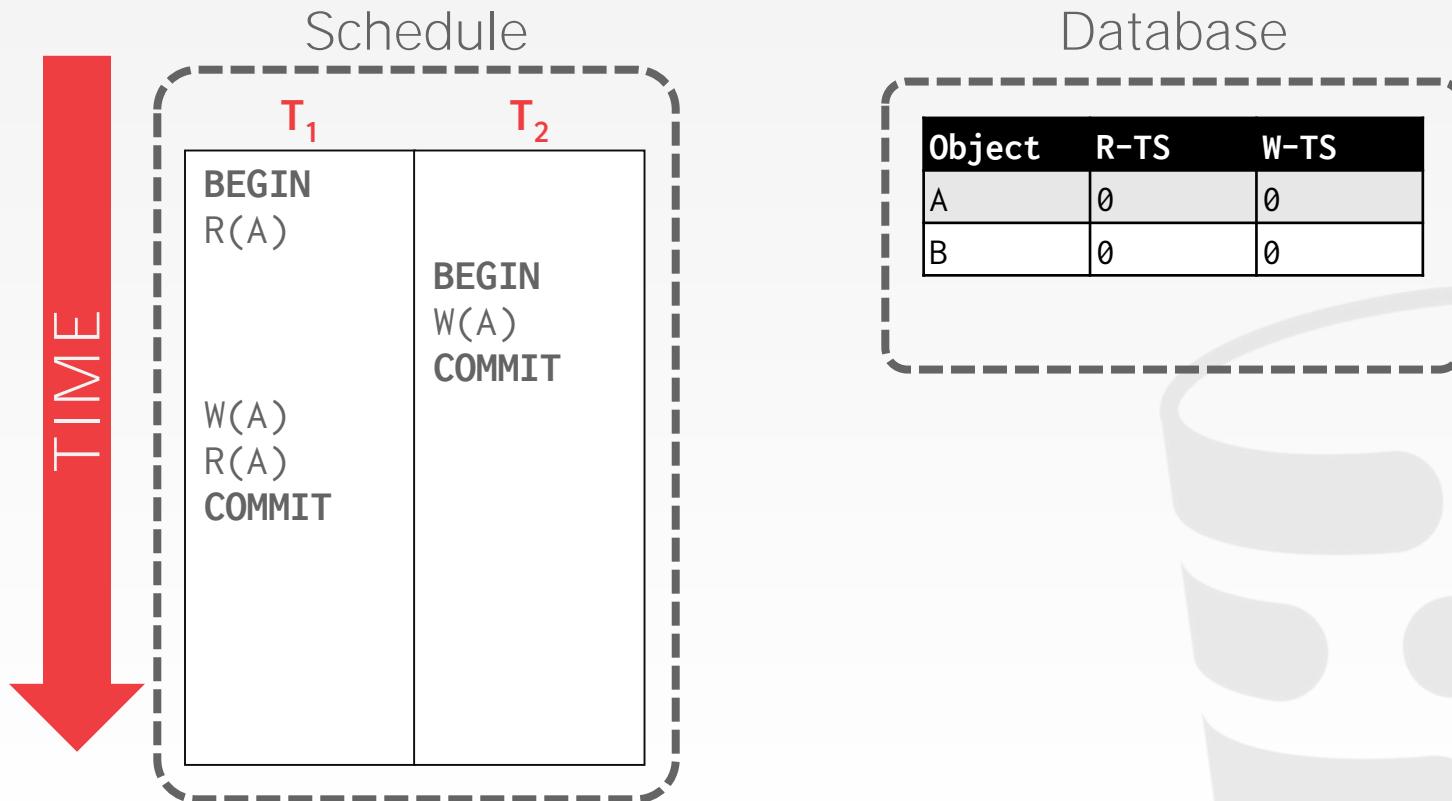
BASIC T/O - EXAMPLE #1



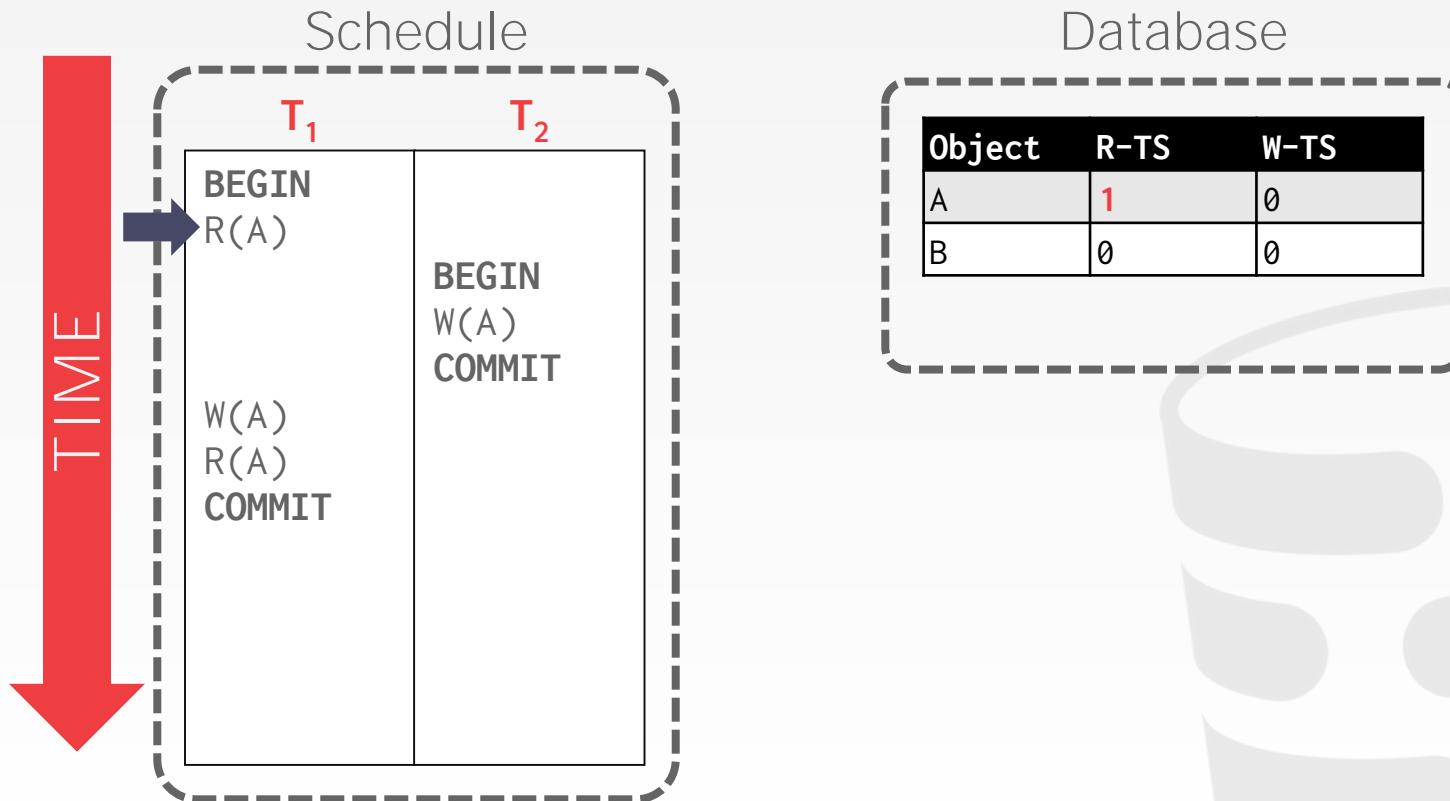
BASIC T/O - EXAMPLE #1



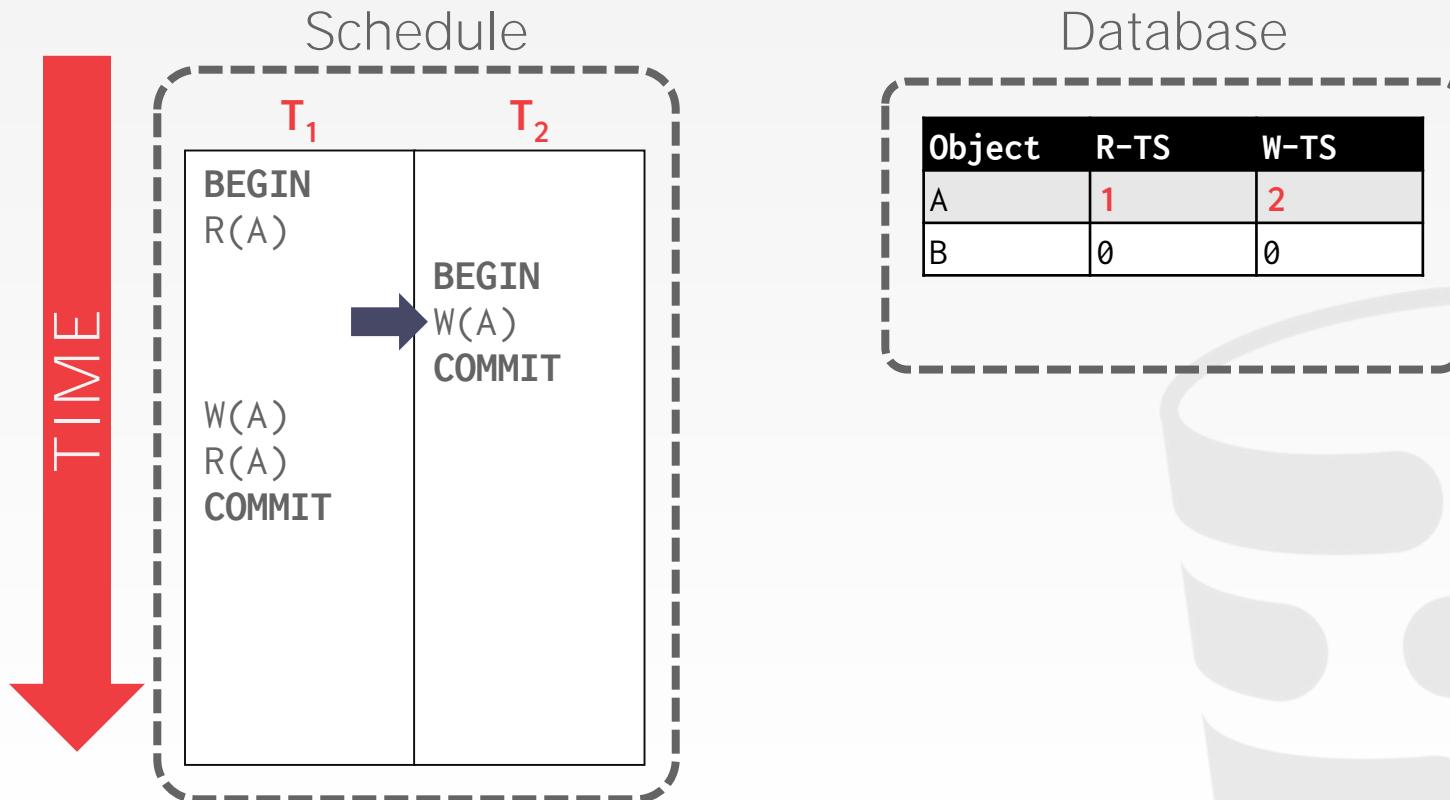
BASIC T/O - EXAMPLE #2



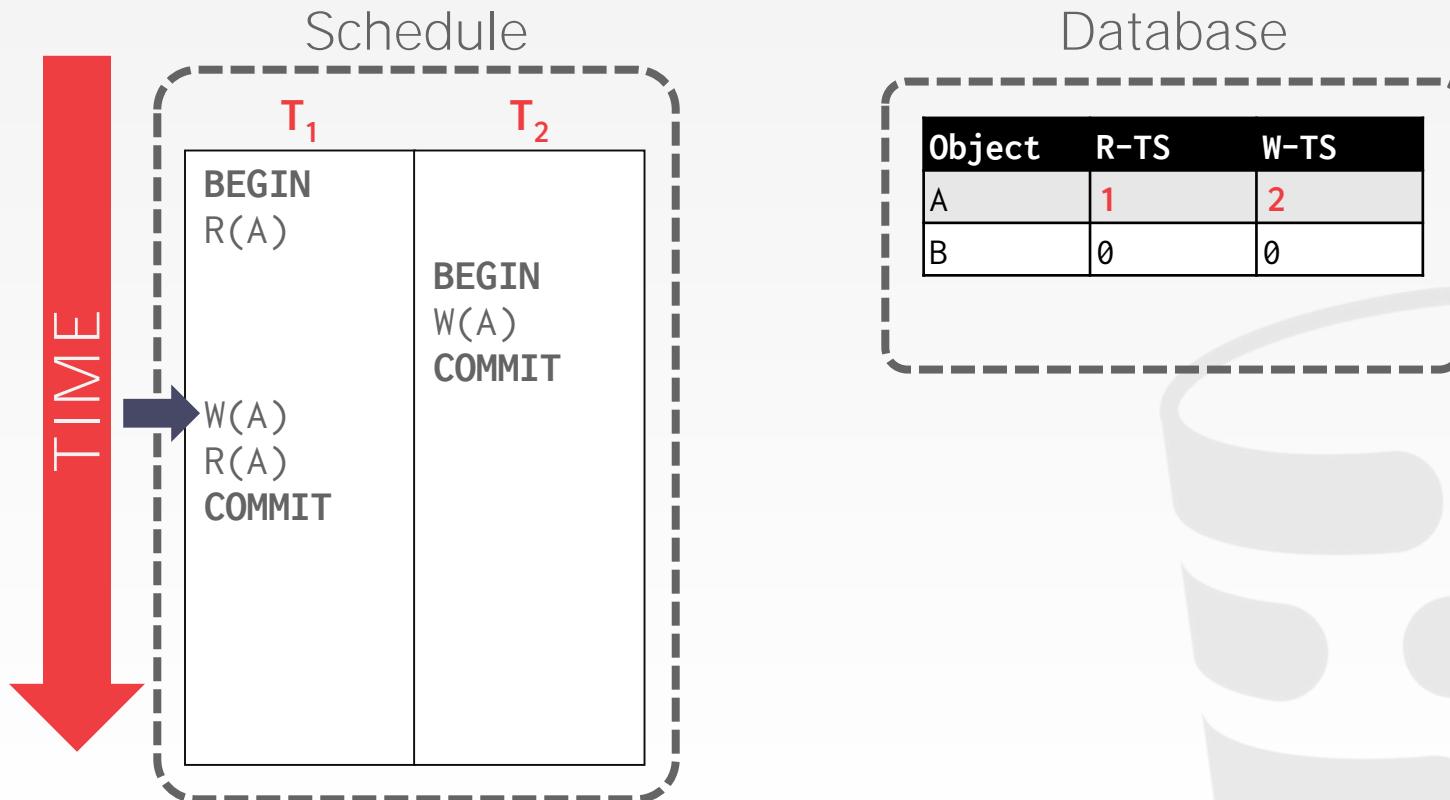
BASIC T/O - EXAMPLE #2



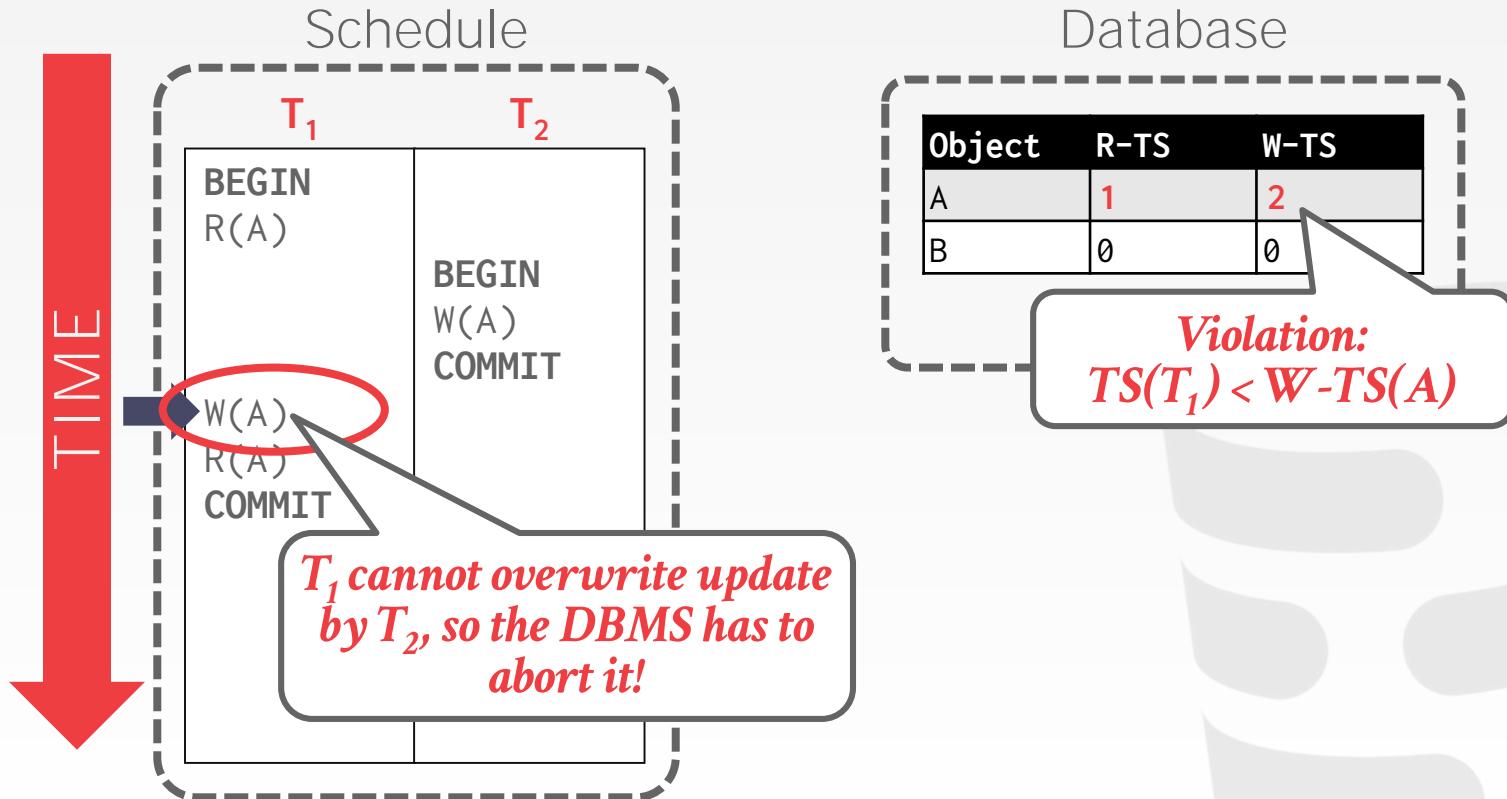
BASIC T/O - EXAMPLE #2



BASIC T/O - EXAMPLE #2



BASIC T/O – EXAMPLE #2



THOMAS WRITE RULE

If $TS(T_i) < R-TS(X)$:

→ Abort and restart T_i .

If $TS(T_i) < W-TS(X)$:

→ Thomas Write Rule: Ignore the write and allow the txn to continue.

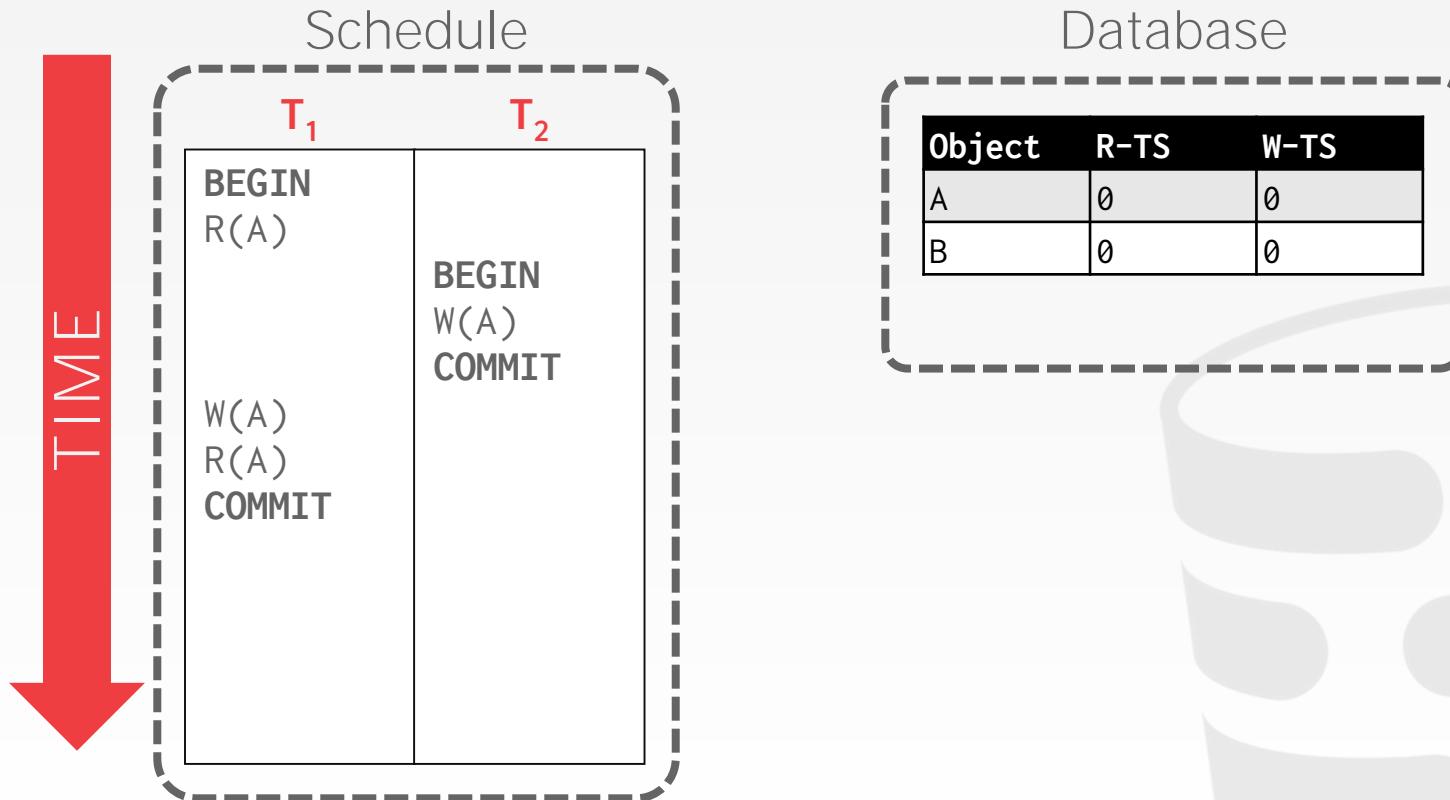
→ This violates timestamp order of T_i .

Else:

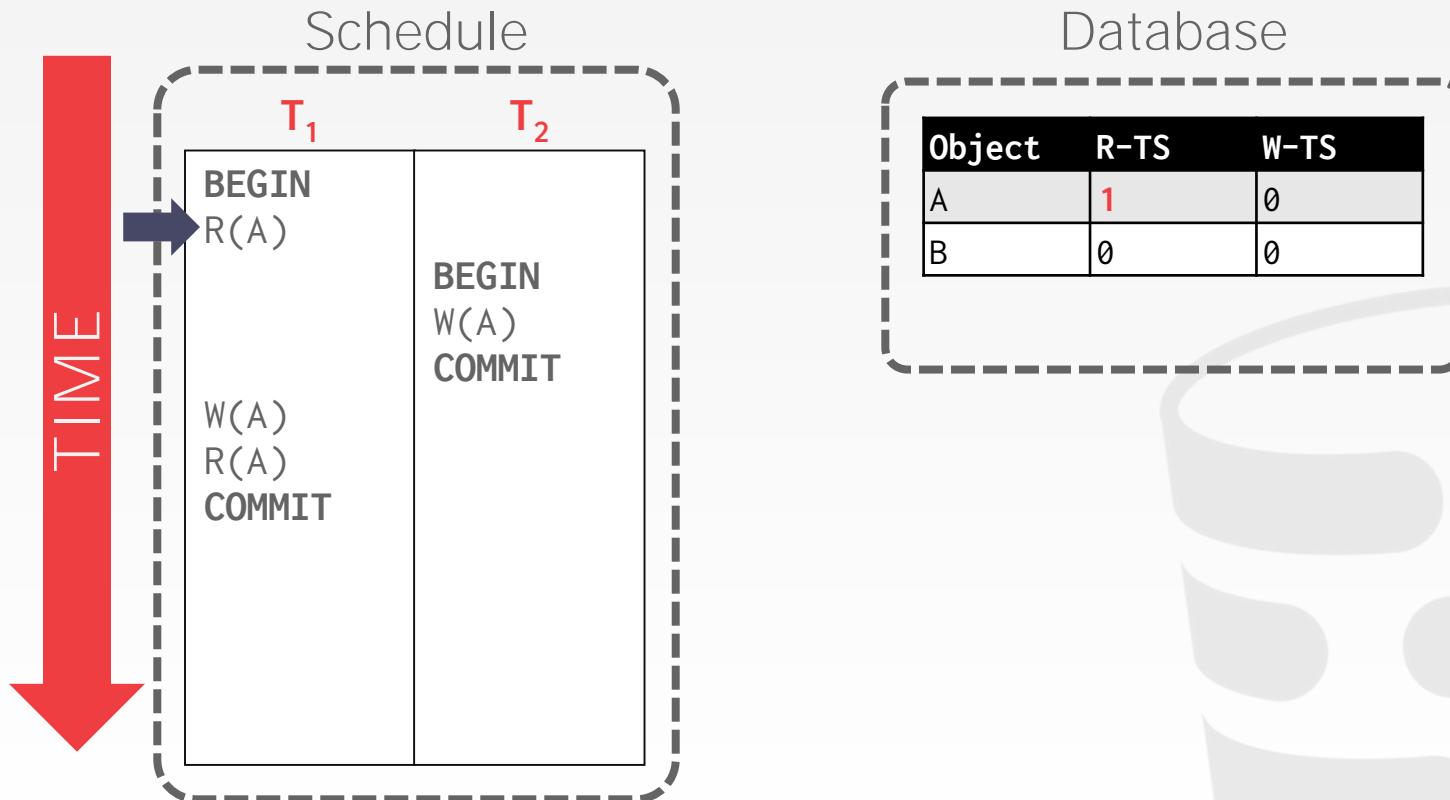
→ Allow T_i to write X and update $W-TS(X)$



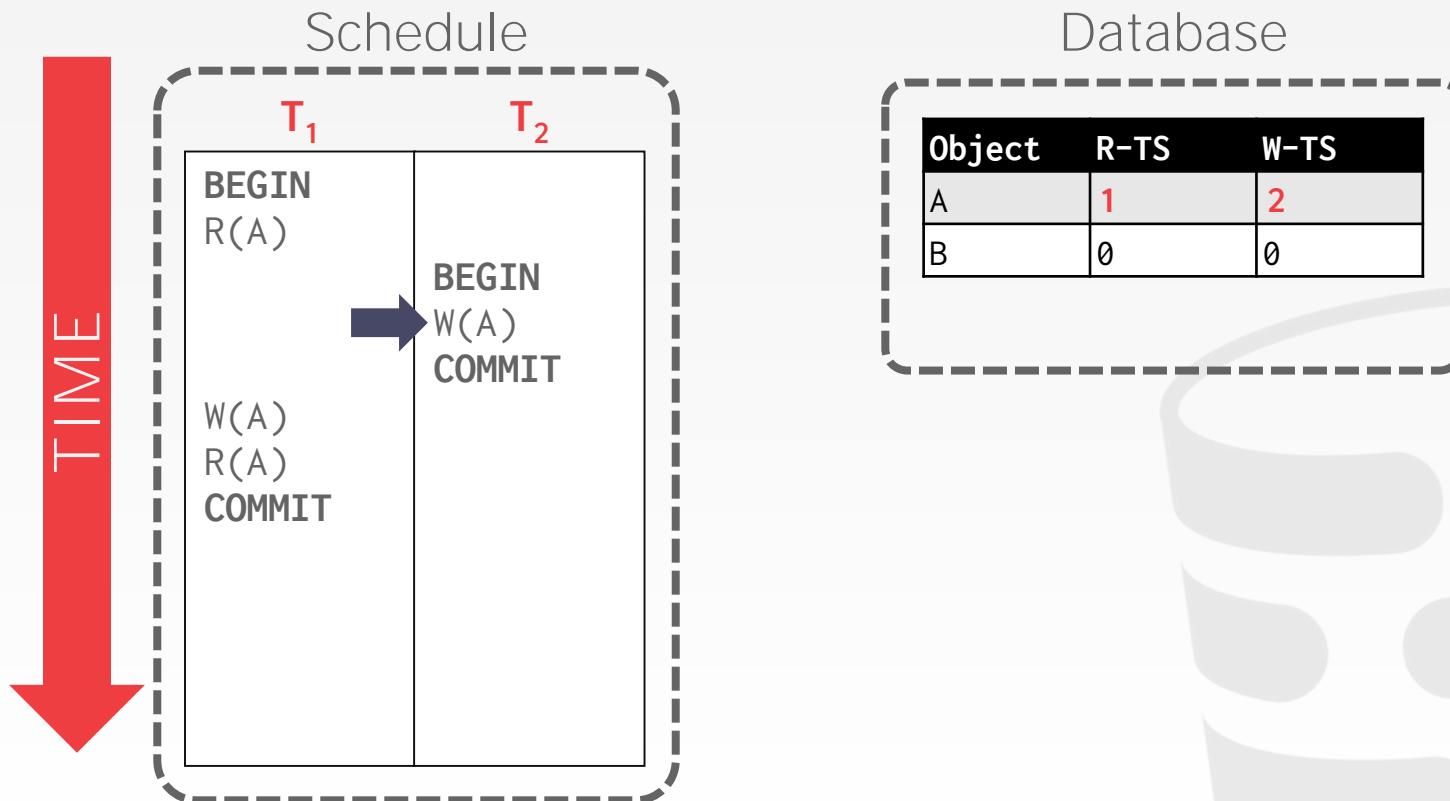
BASIC T/O - EXAMPLE #2



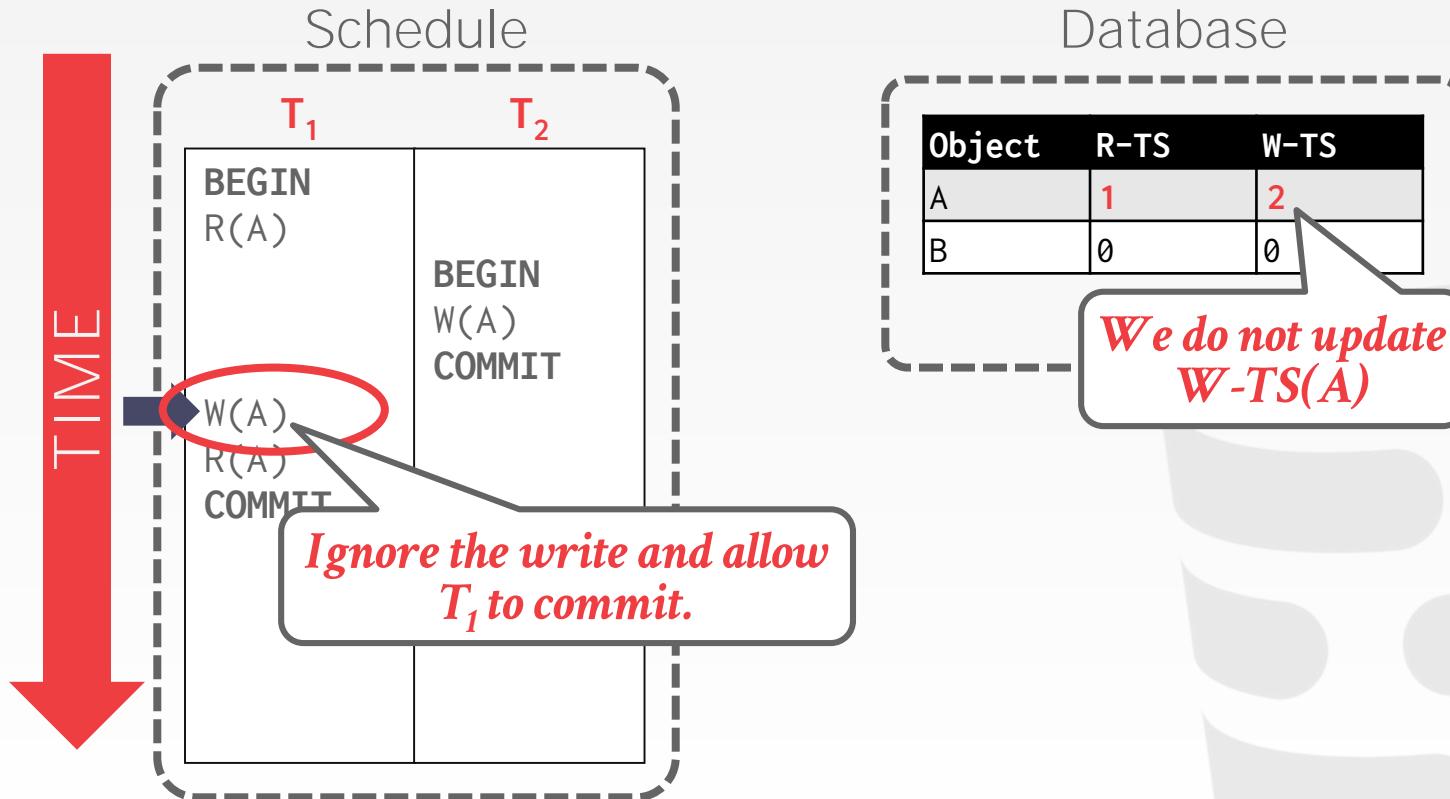
BASIC T/O - EXAMPLE #2



BASIC T/O - EXAMPLE #2



BASIC T/O – EXAMPLE #2



BASIC T/O

Generates a schedule that is conflict serializable if you do not use the Thomas Write Rule.

- No deadlocks because no txn ever waits.
- Possibility of starvation for long txns if short txns keep causing conflicts.

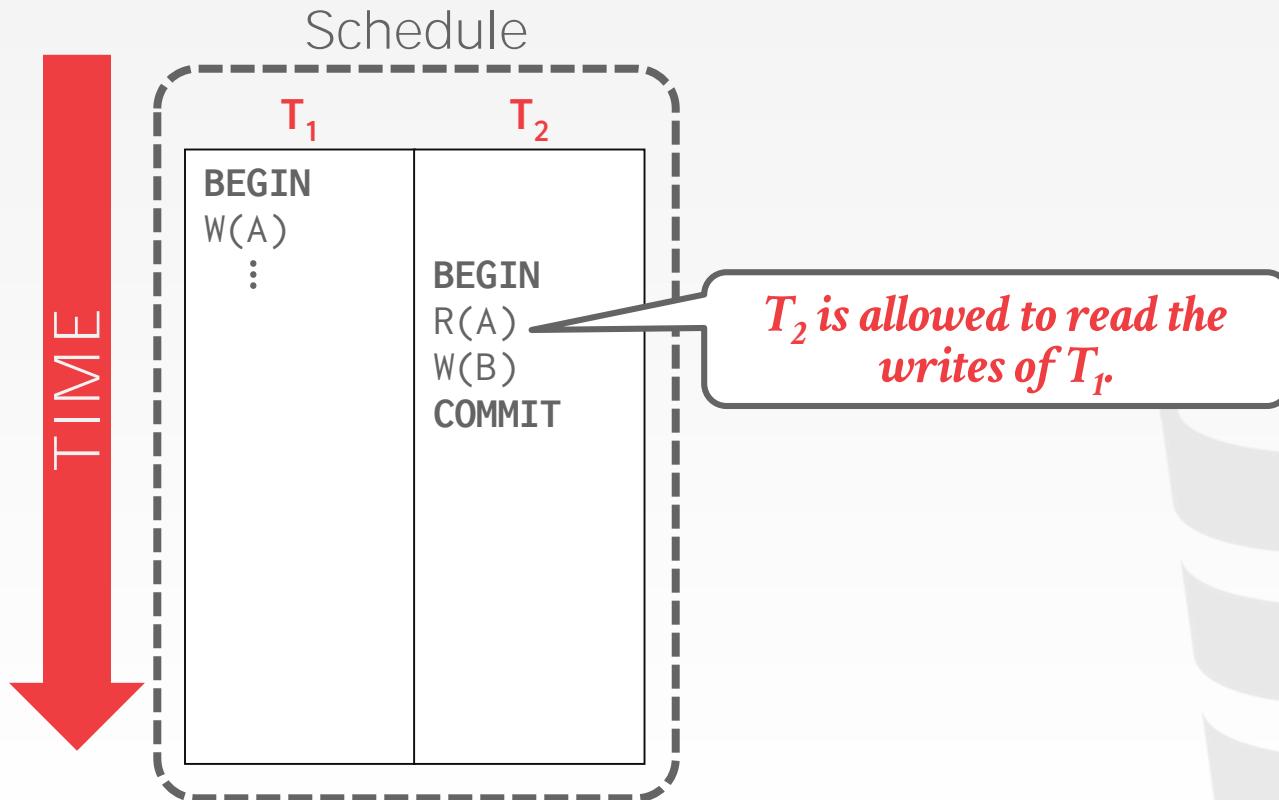
Permits schedules that are not recoverable.

RECOVERABLE SCHEDULES

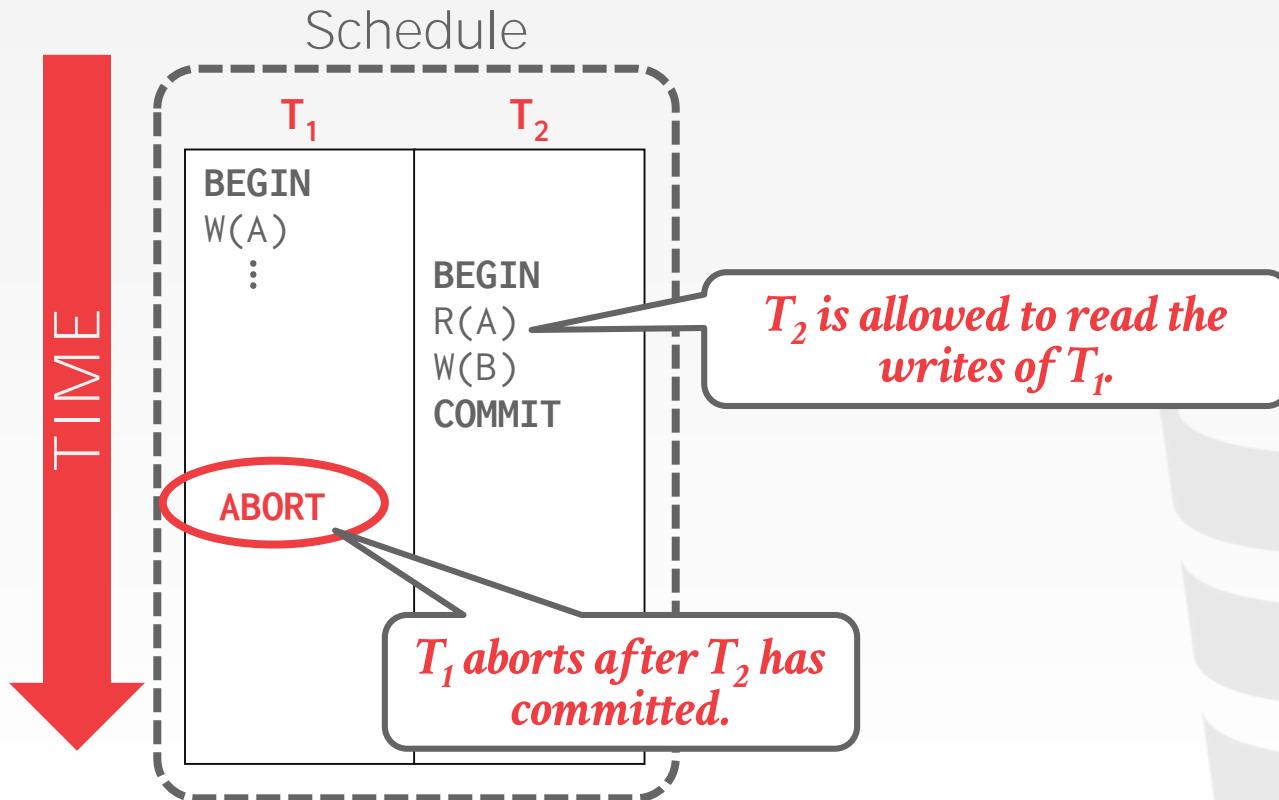
A schedule is **recoverable** if txns commit only after all txns whose changes they read, commit.

Otherwise, the DBMS cannot guarantee that txns read data that will be restored after recovering from a crash.

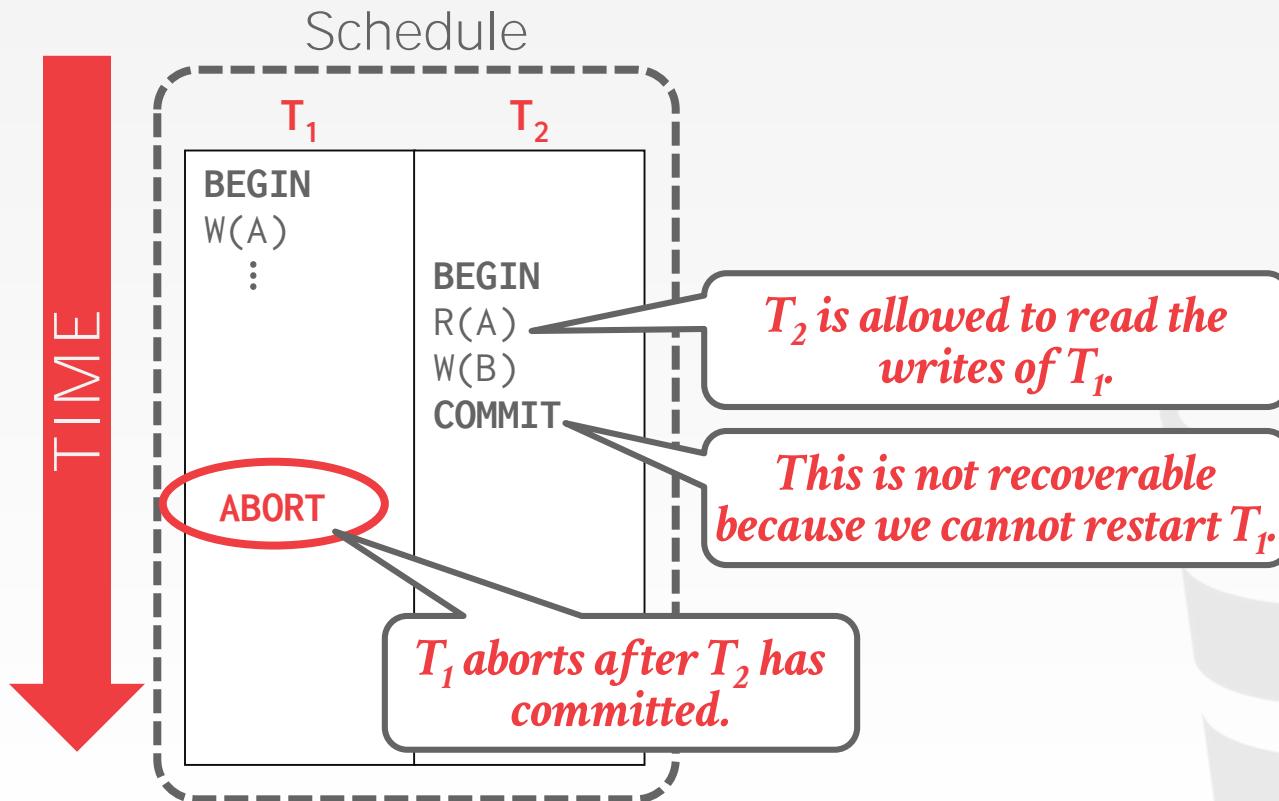
RECOVERABLE SCHEDULES



RECOVERABLE SCHEDULES



RECOVERABLE SCHEDULES



BASIC T/O – PERFORMANCE ISSUES

High overhead from copying data to txn's workspace and from updating timestamps.

Long running txns can get starved.

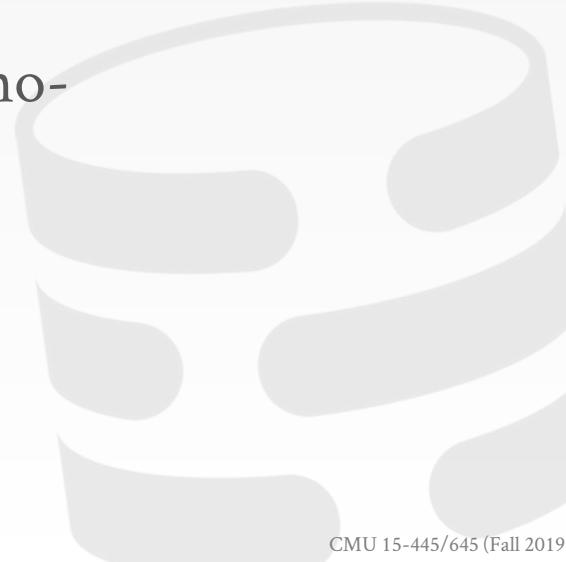
→ The likelihood that a txn will read something from a newer txn increases.



OBSERVATION

If you assume that conflicts between txns are rare and that most txns are short-lived, then forcing txns to wait to acquire locks adds a lot of overhead.

A better approach is to optimize for the no-conflict case.



OPTIMISTIC CONCURRENCY CONTROL

The DBMS creates a private workspace for each txn.

- Any object read is copied into workspace.
- Modifications are applied to workspace.

When a txn commits, the DBMS compares workspace write set to see whether it conflicts with other txns.

If there are no conflicts, the write set is installed into the "global" database.

On Optimistic Methods for Concurrency Control

H.T. KUNG and JOHN T. ROBINSON
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

Key Words and Phrases: databases, concurrency controls, transaction processing
CR Categories: 4.32, 4.33

1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

- (1) The amount of data is sufficiently great that at any given time only a fraction of the database can be present in primary memory, so that it is necessary to swap parts of the database from secondary memory as needed.
- (2) Even if the entire database can be present in primary memory, there may be multiple processors.

In both cases the hardware will be underutilized if the degree of concurrency is too low.

However, as is well known, unrestricted concurrent access to a shared database will, in general, cause the integrity of the database to be lost. Most current

Permission to copy without fee for all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0373.
Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.
© 1981 ACM 0892-3915/81/0600-0213 \$00.75
ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, Pages 213-226

OCC PHASES

#1 – Read Phase:

- Track the read/write sets of txns and store their writes in a private workspace.

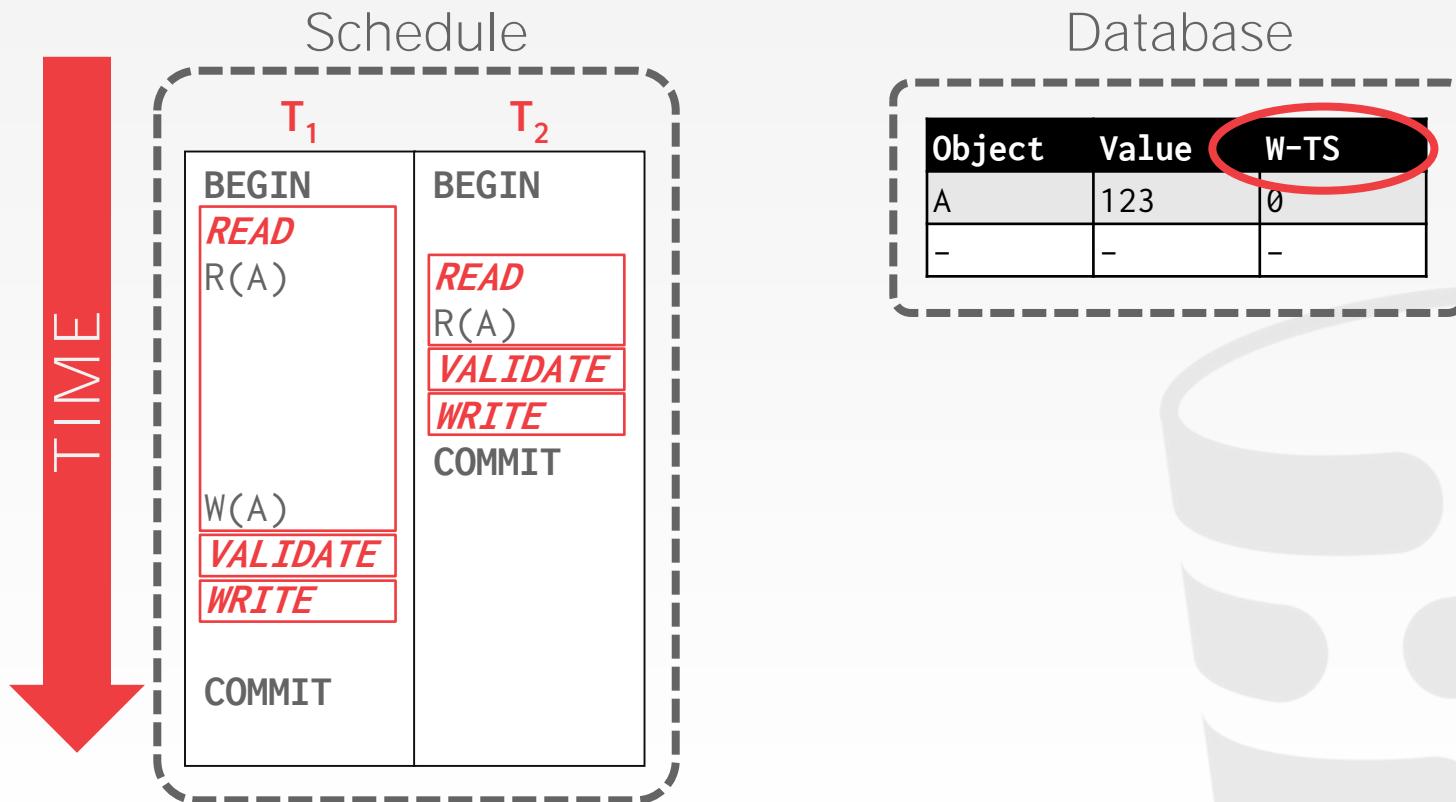
#2 – Validation Phase:

- When a txn commits, check whether it conflicts with other txns.

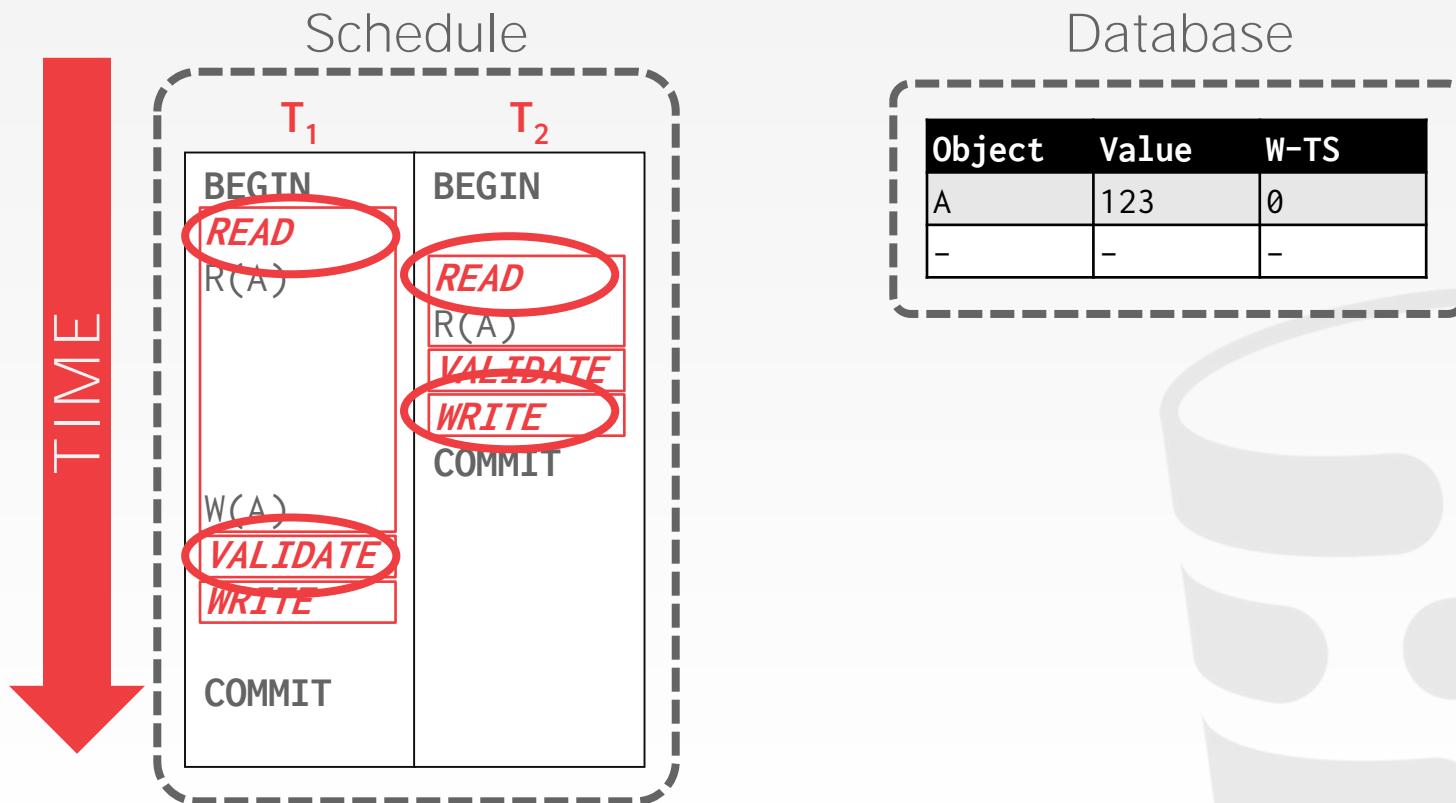
#3 – Write Phase:

- If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.

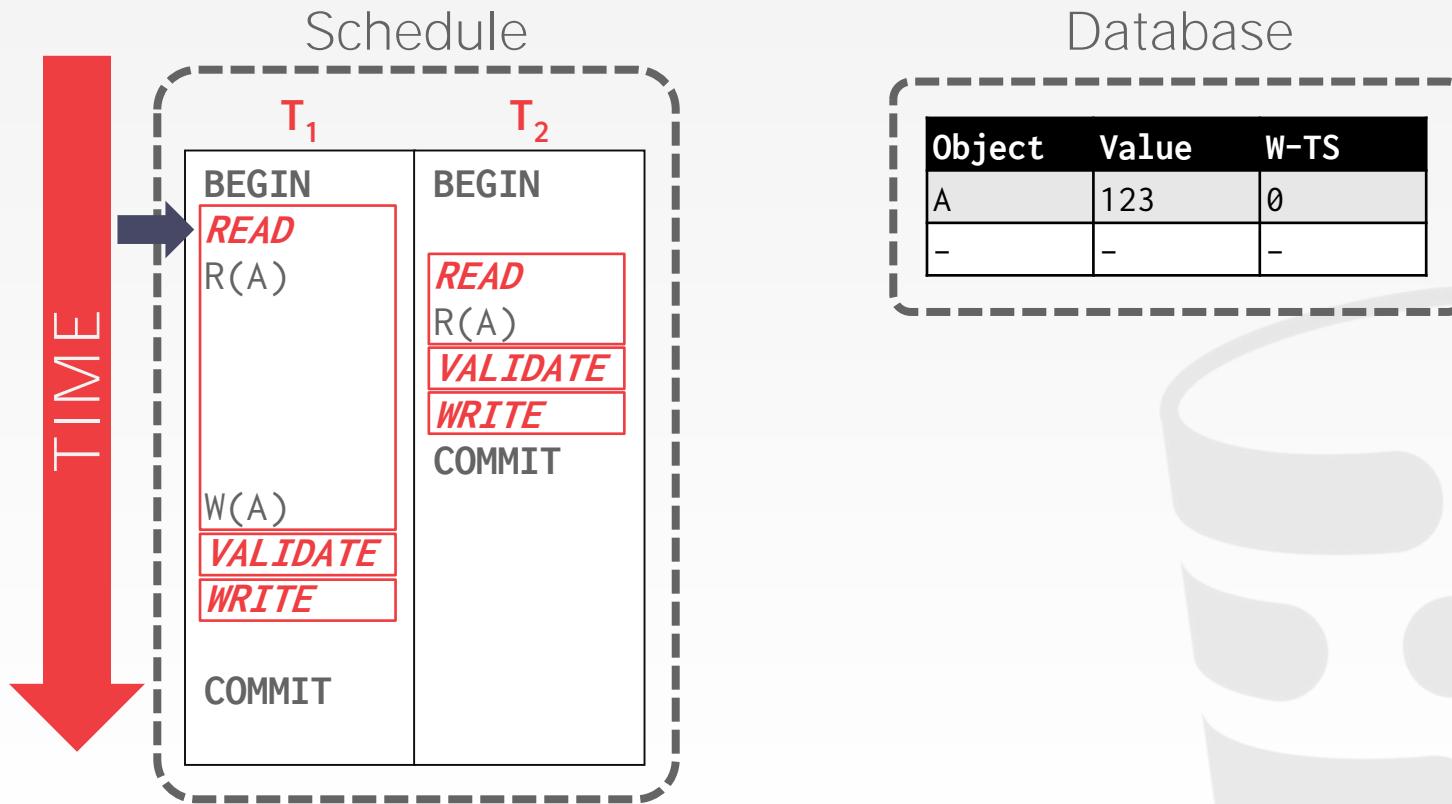
OCC – EXAMPLE



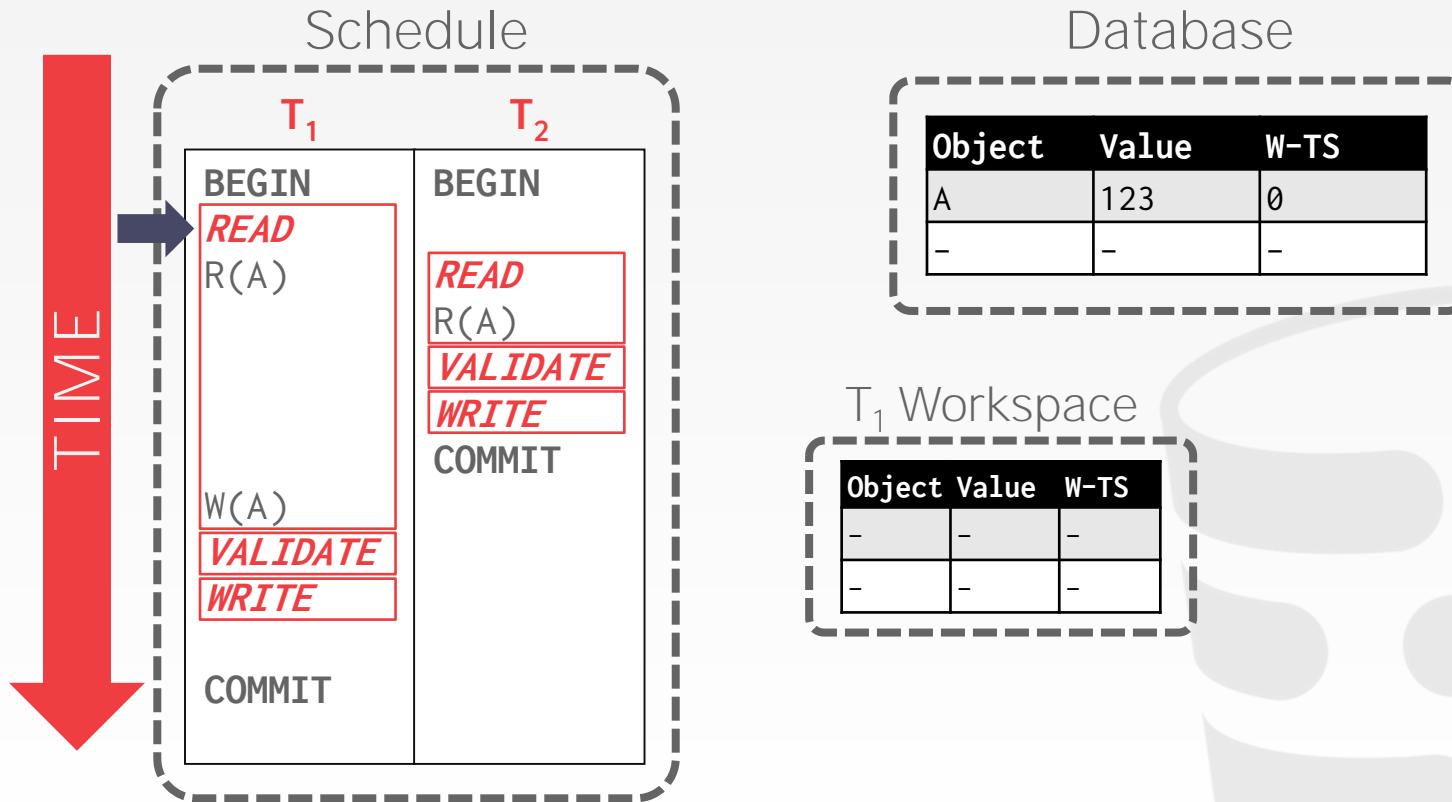
OCC – EXAMPLE



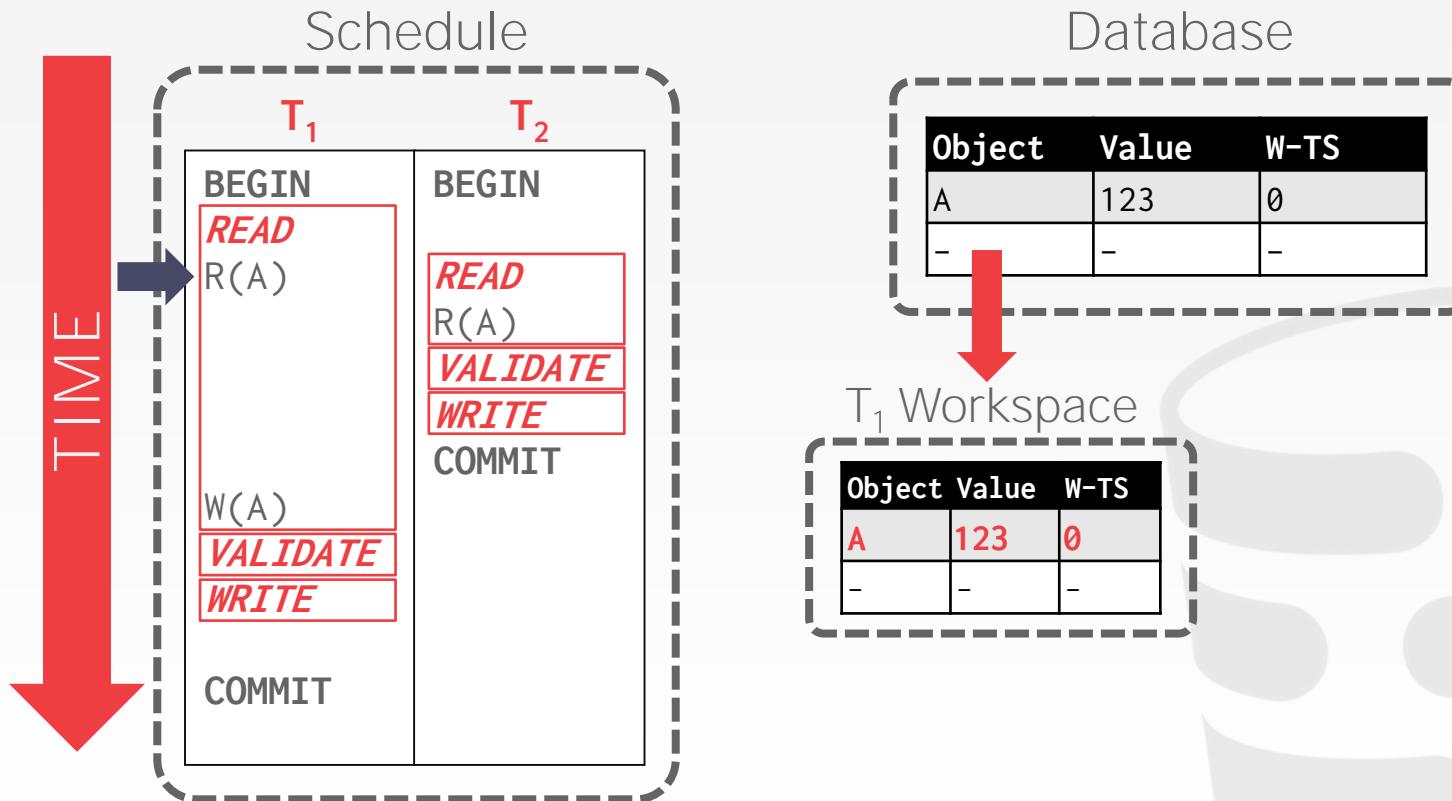
OCC – EXAMPLE



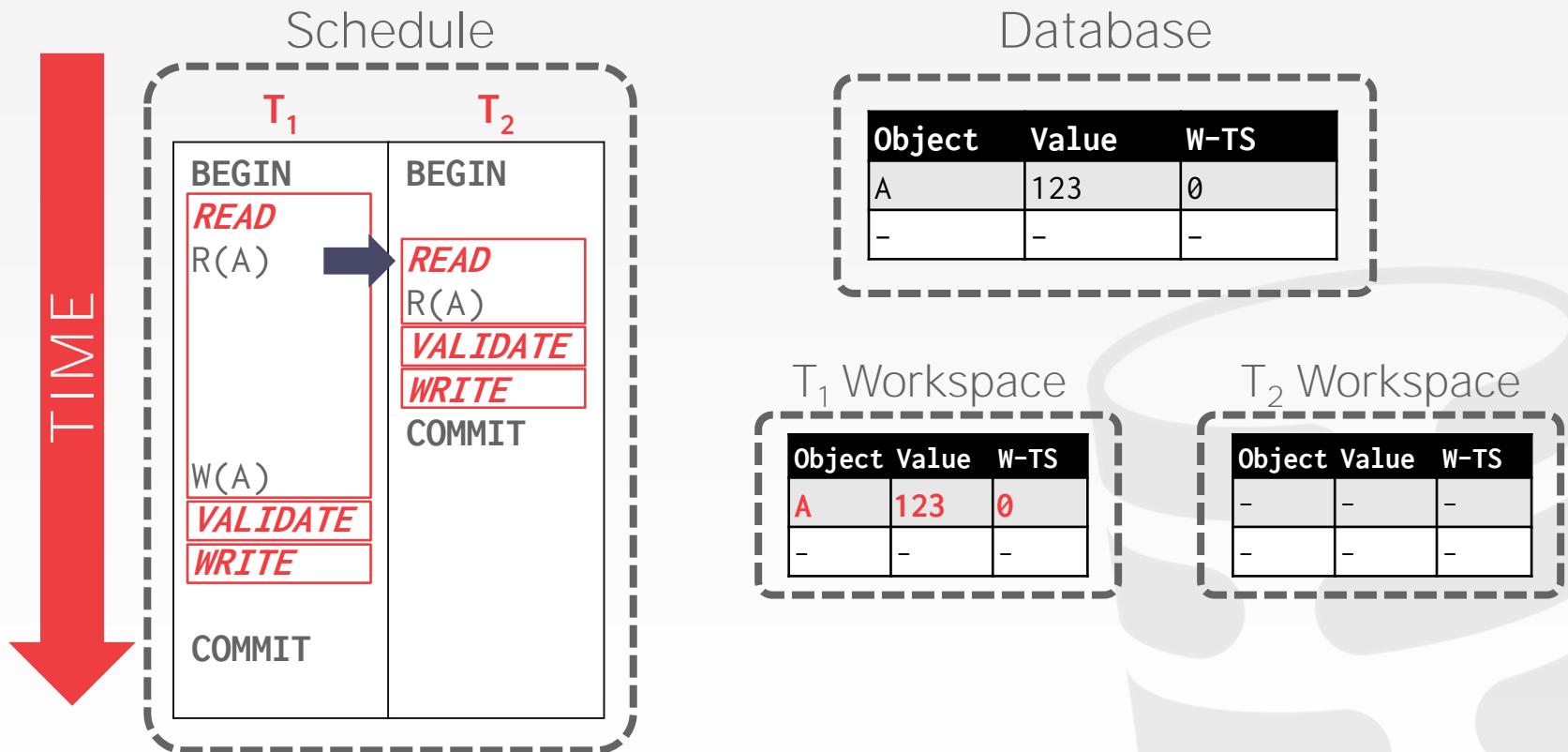
OCC – EXAMPLE



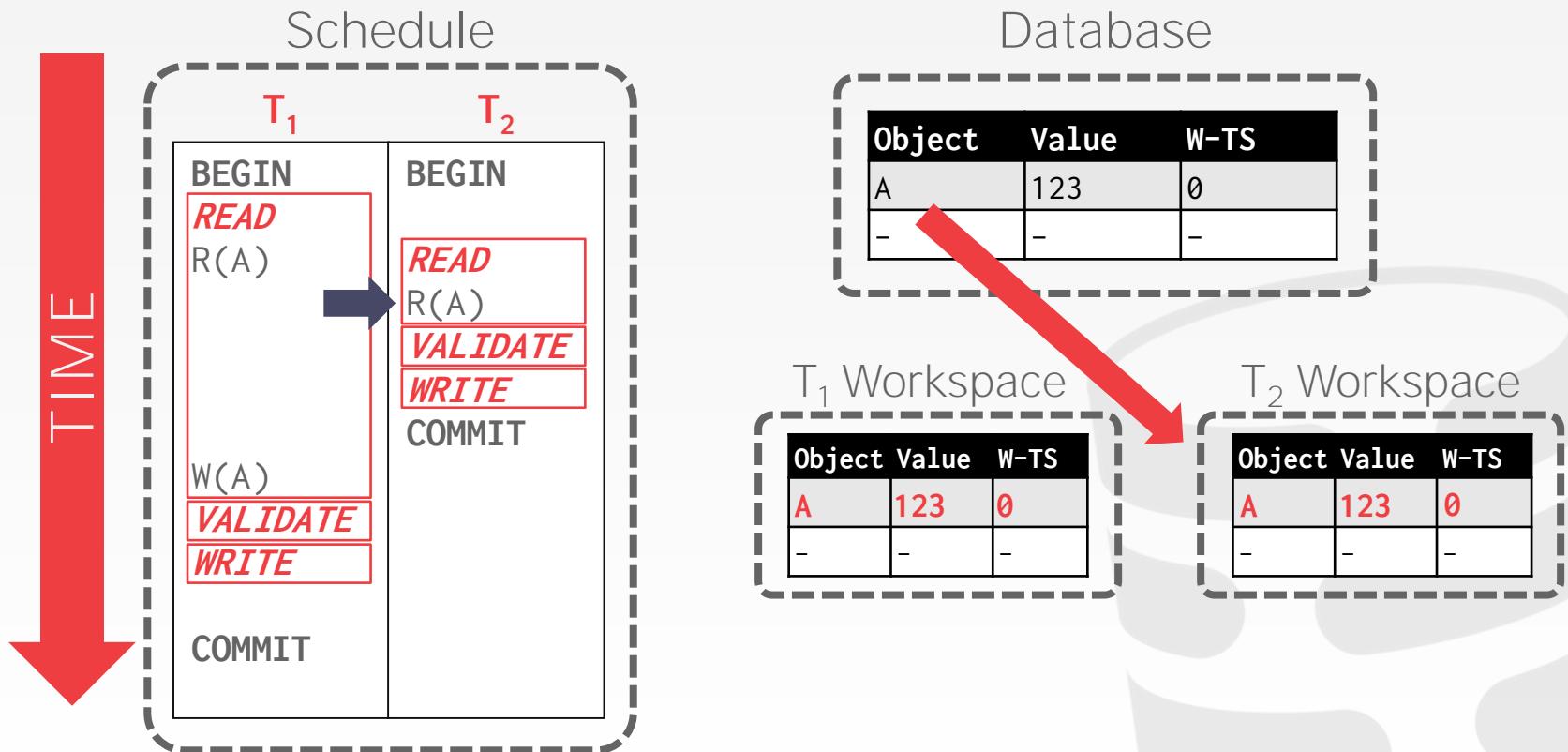
OCC – EXAMPLE



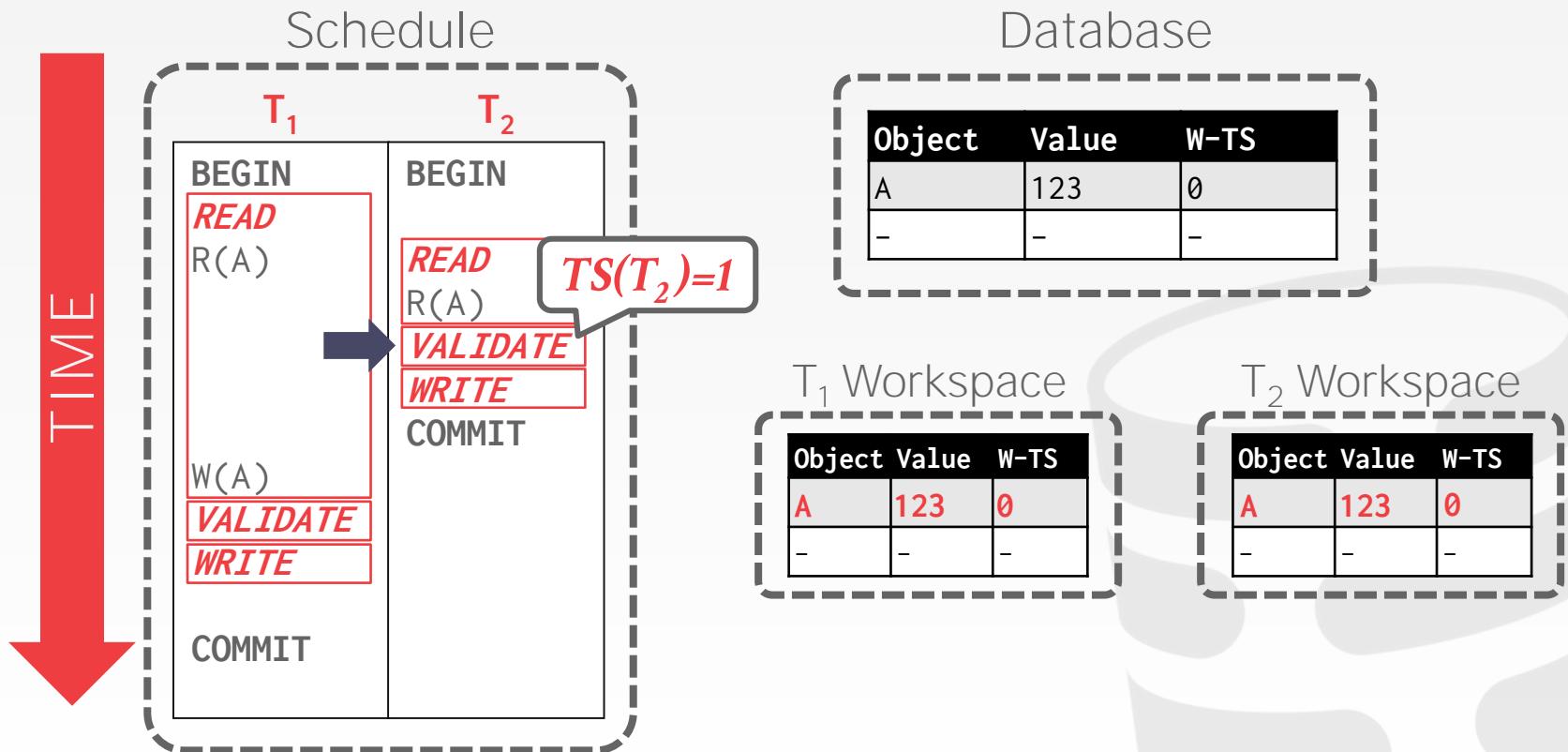
OCC – EXAMPLE



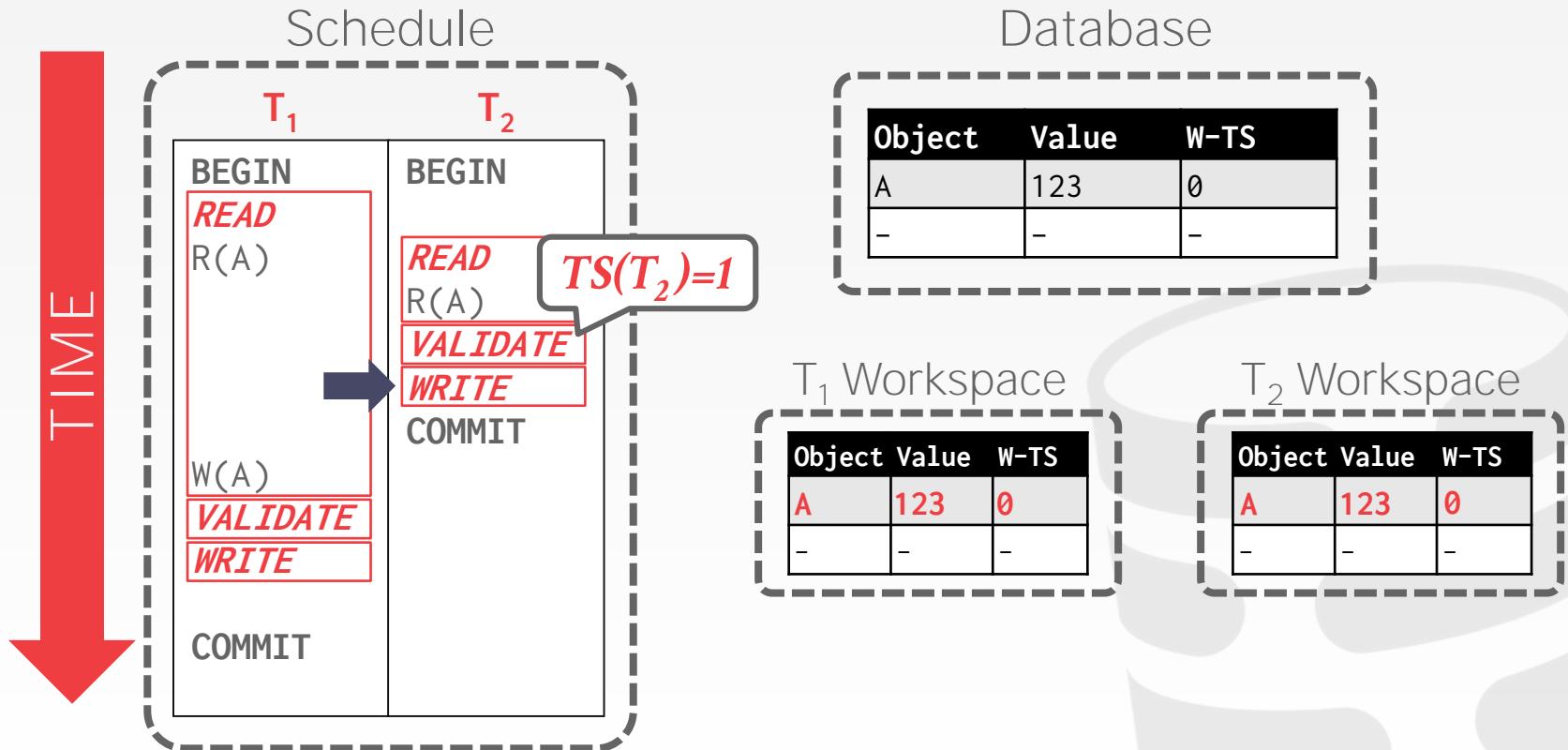
OCC – EXAMPLE



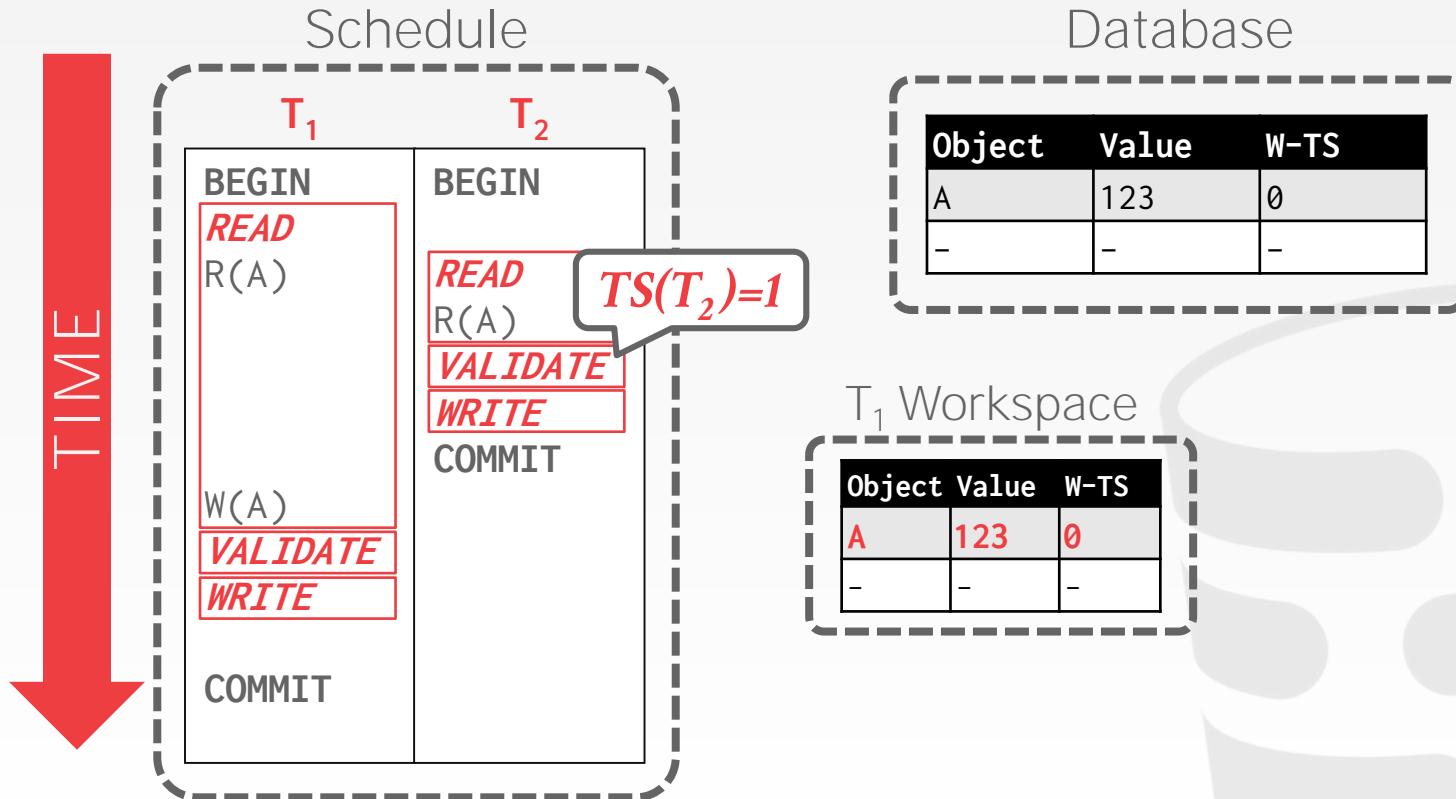
OCC – EXAMPLE



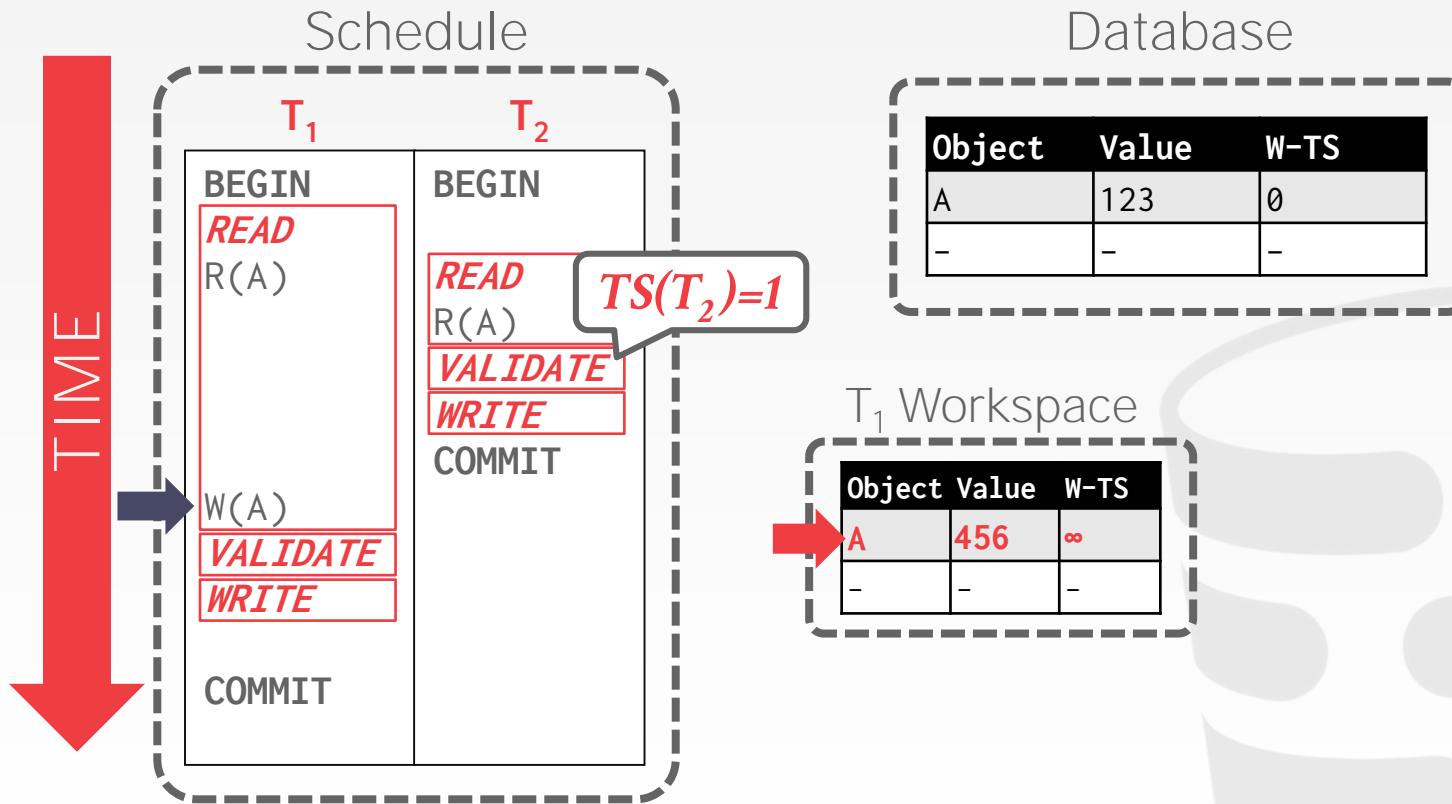
OCC – EXAMPLE



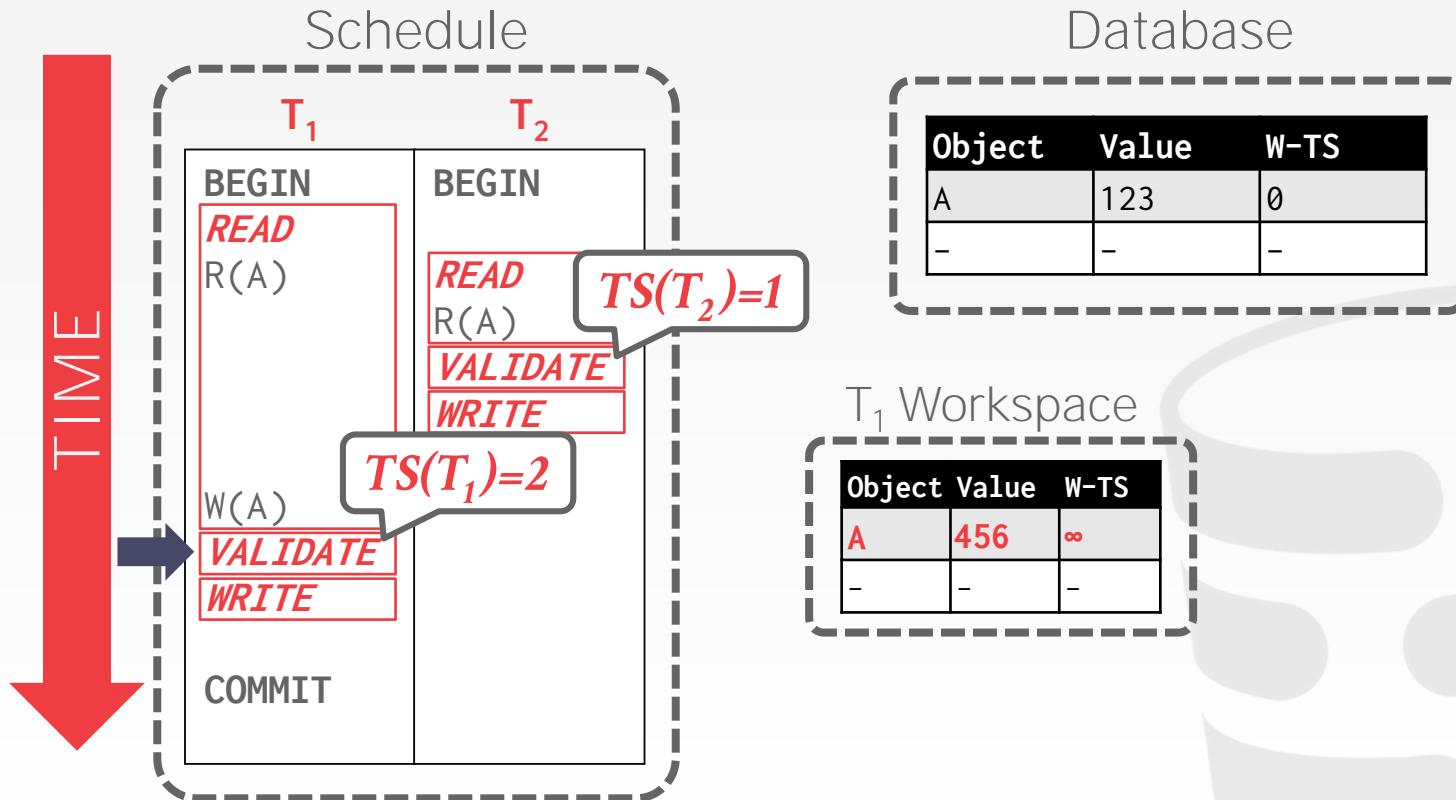
OCC – EXAMPLE



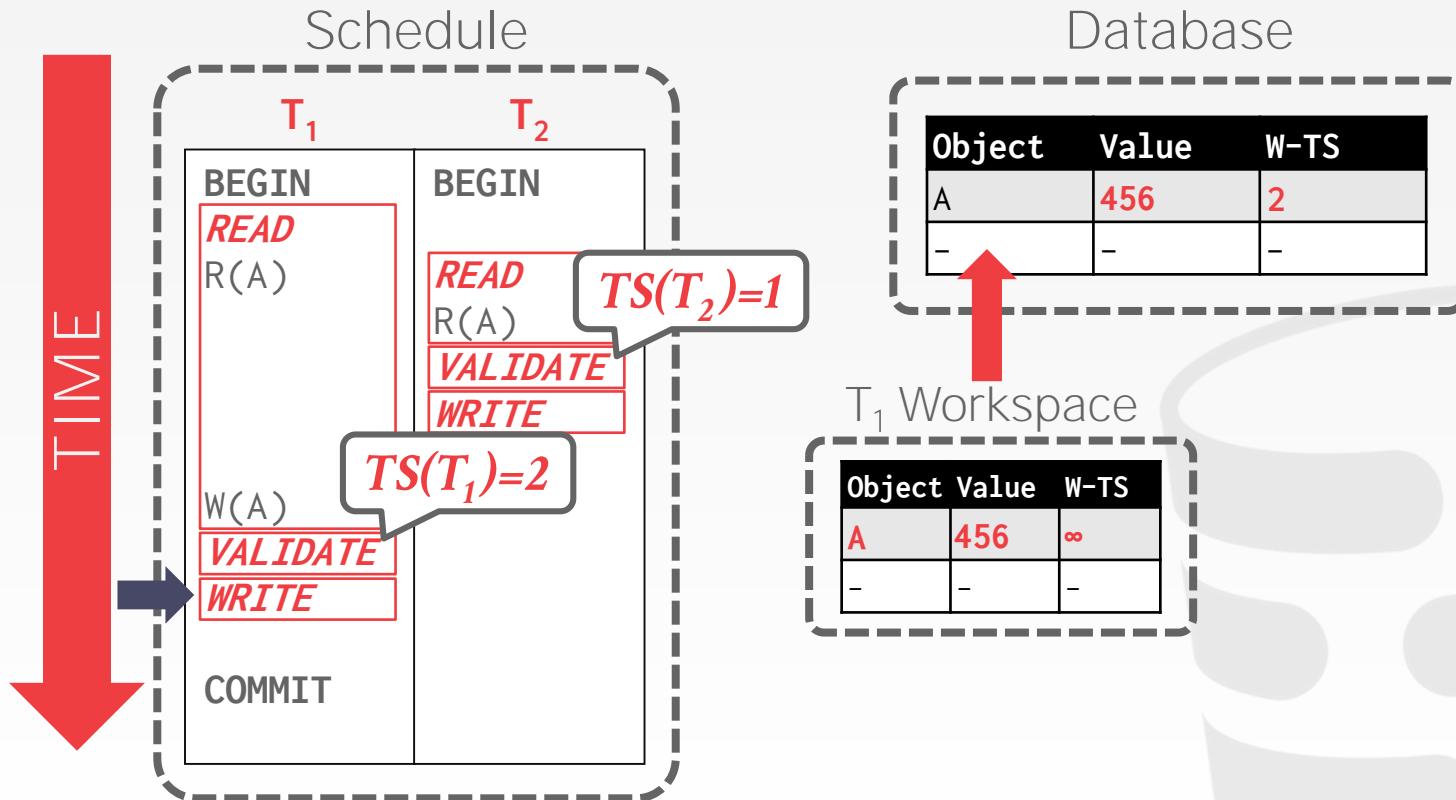
OCC – EXAMPLE



OCC – EXAMPLE



OCC – EXAMPLE



OCC – VALIDATION PHASE

The DBMS needs to guarantee only serializable schedules are permitted.

T_i checks other txns for RW and WW conflicts and makes sure that all conflicts go one way (from older txns to younger txns).



OCC – SERIAL VALIDATION

Maintain global view of all active txns.

Record read set and write set while txns are running and write into private workspace.

Execute **Validation** and **Write** phase inside a protected critical section.

OCC – READ PHASE

Track the read/write sets of txns and store their writes in a private workspace.

The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.



OCC – VALIDATION PHASE

Each txn's timestamp is assigned at the beginning of the validation phase.

Check the timestamp ordering of the committing txn with all other running txns.

If $\text{TS}(T_i) < \text{TS}(T_j)$, then one of the following three conditions must hold...



OCC – VALIDATION PHASE

When the txn invokes **COMMIT**, the DBMS checks if it conflicts with other txns.

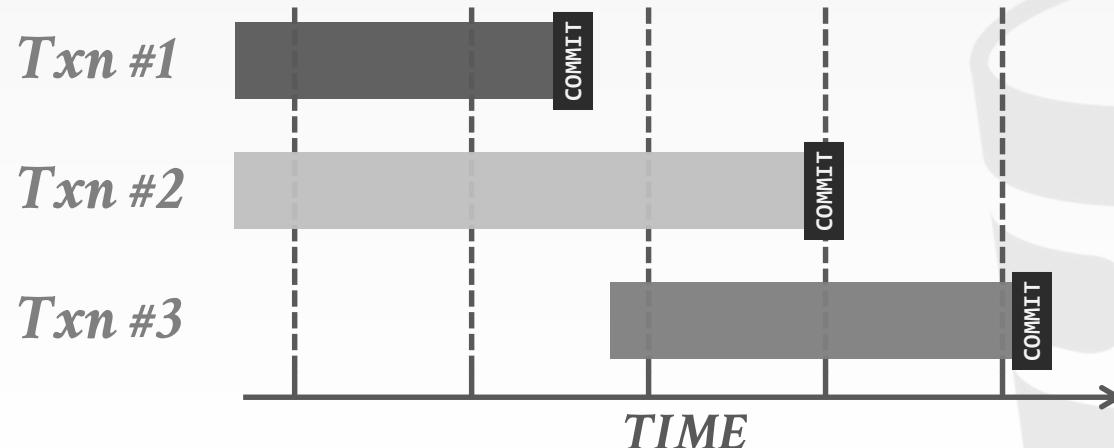
Two methods for this phase:

- Backward Validation
- Forward Validation



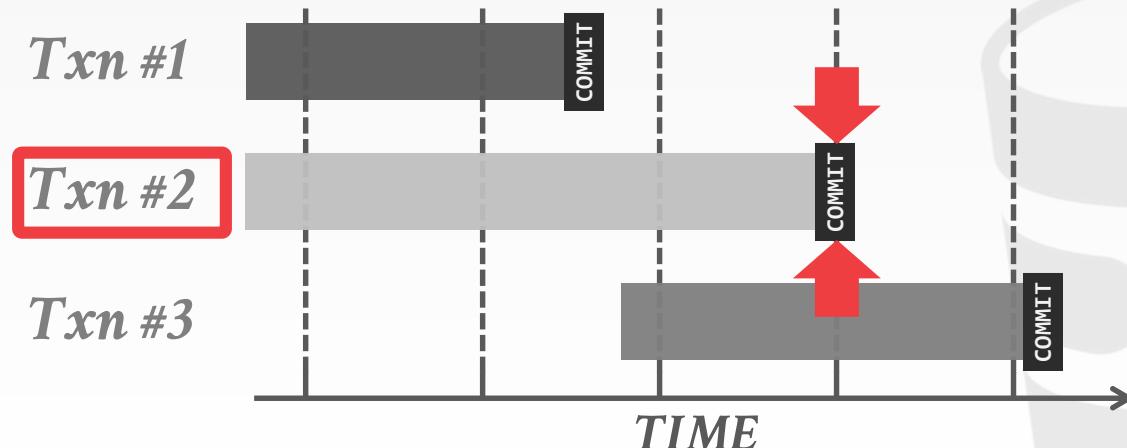
OCC – BACKWARD VALIDATION

Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.



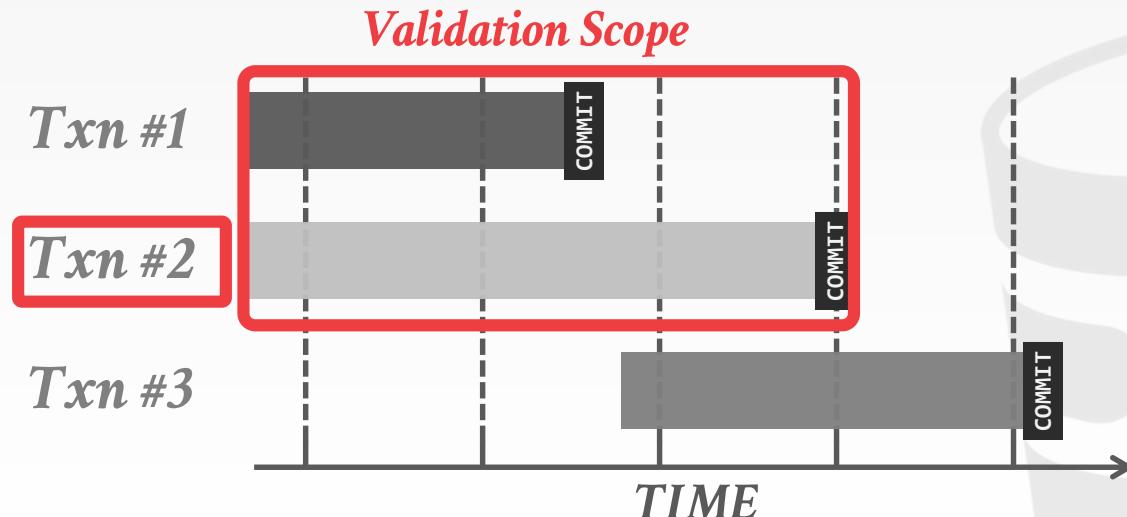
OCC – BACKWARD VALIDATION

Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.



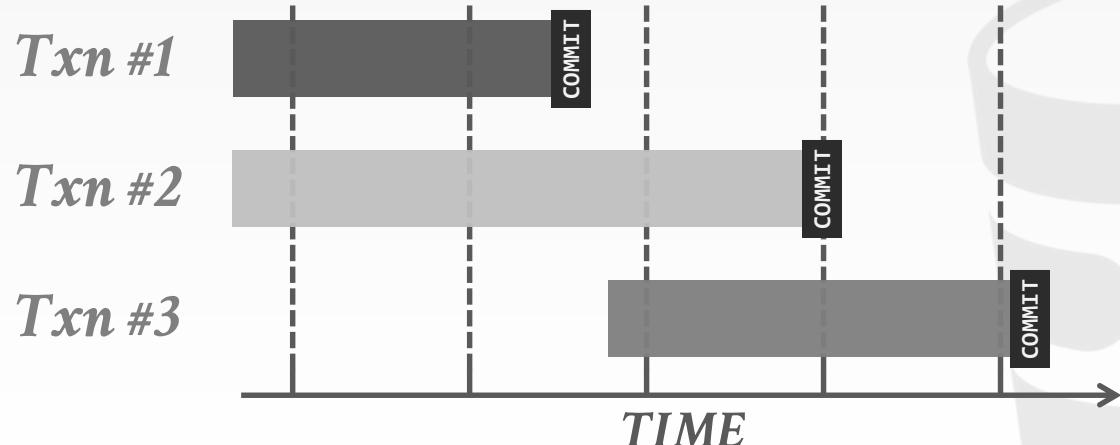
OCC – BACKWARD VALIDATION

Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.



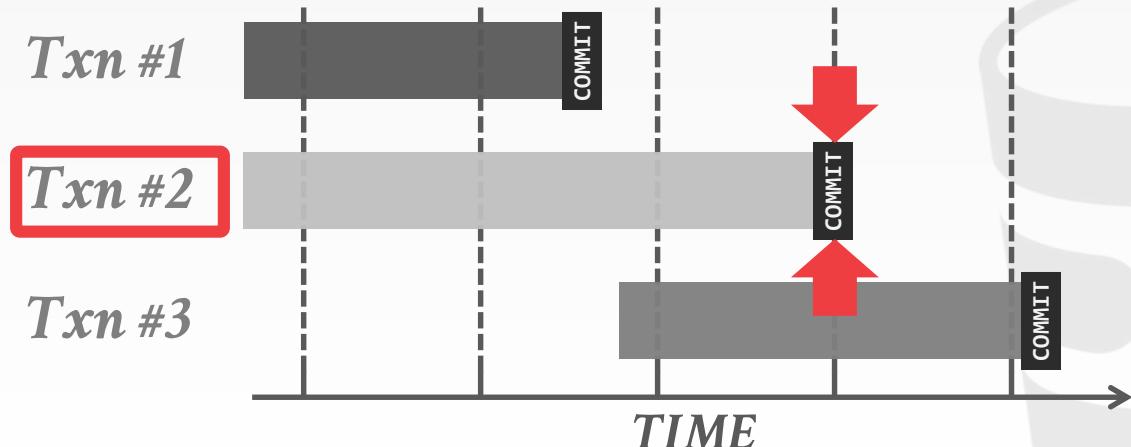
OCC – FORWARD VALIDATION

Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.



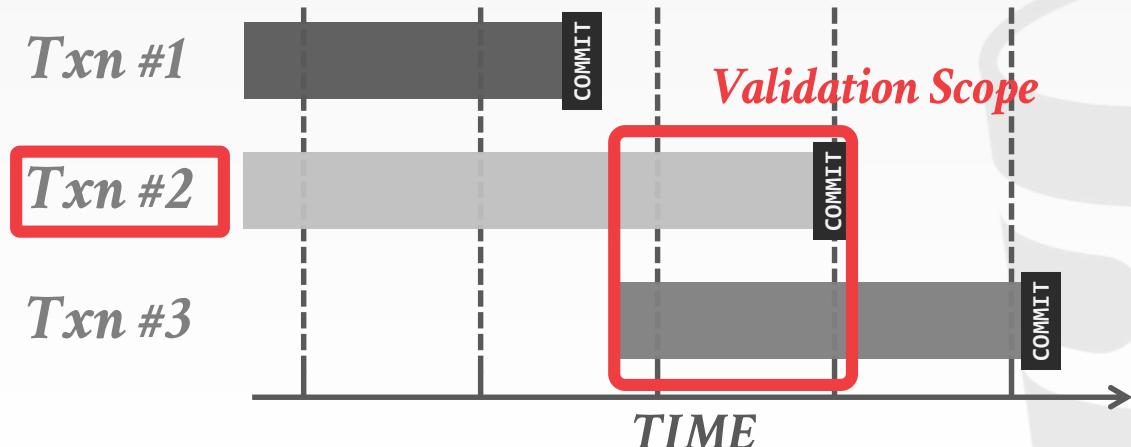
OCC – FORWARD VALIDATION

Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.



OCC – FORWARD VALIDATION

Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.

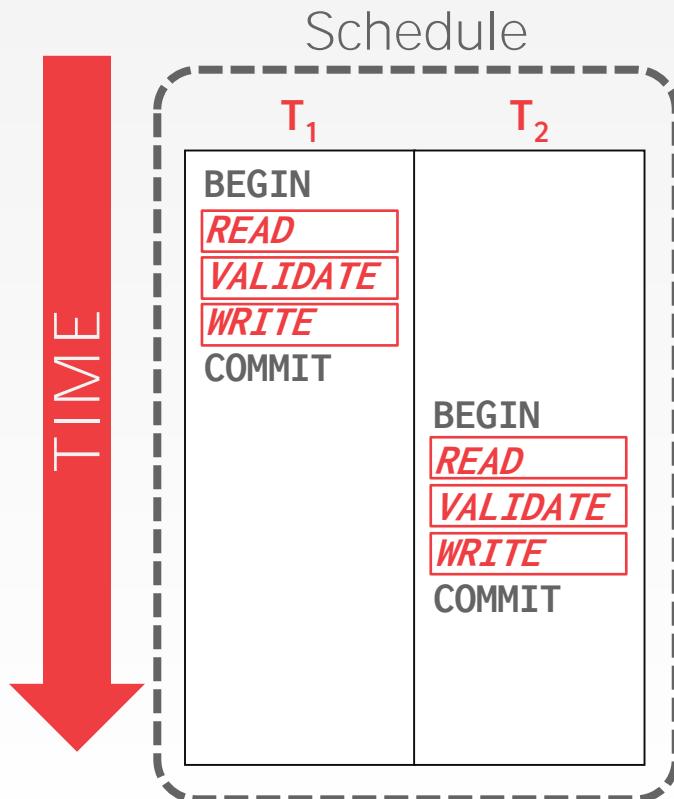


OCC – VALIDATION STEP #1

T_i completes all three phases before T_j begins.



OCC – VALIDATION STEP #1

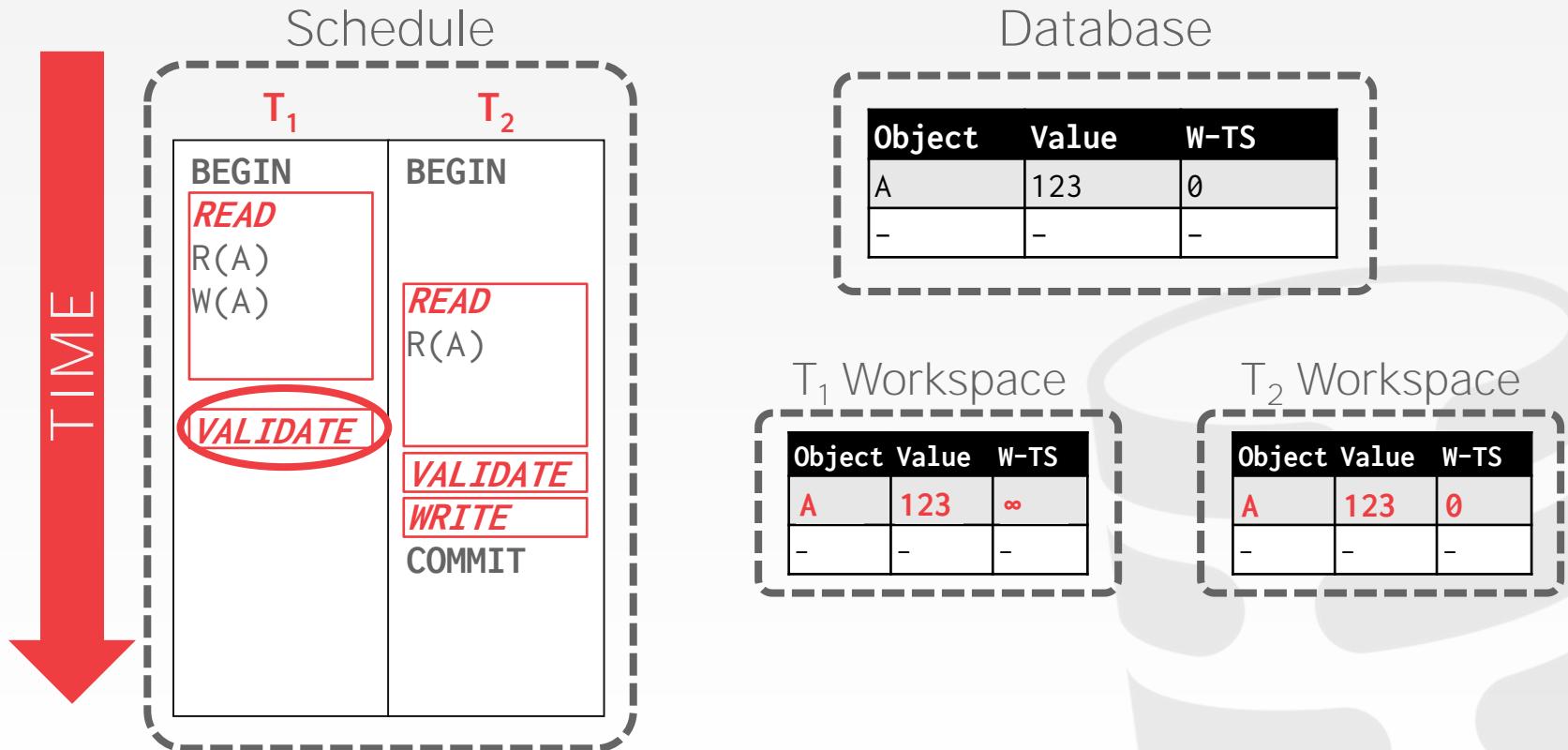


OCC – VALIDATION STEP #2

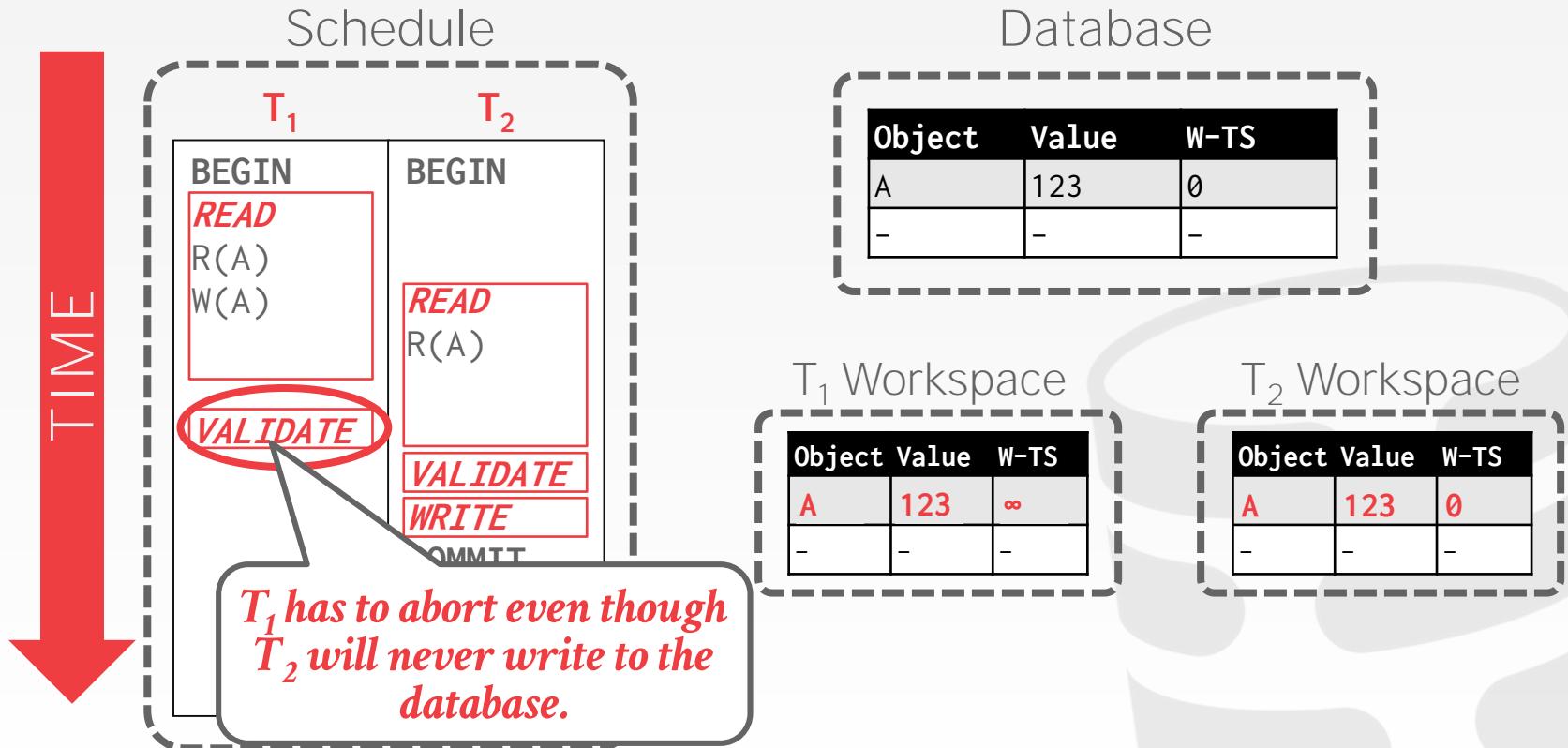
T_i completes before T_j starts its **Write** phase, and
 T_i does not write to any object read by T_j .
→ $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset$



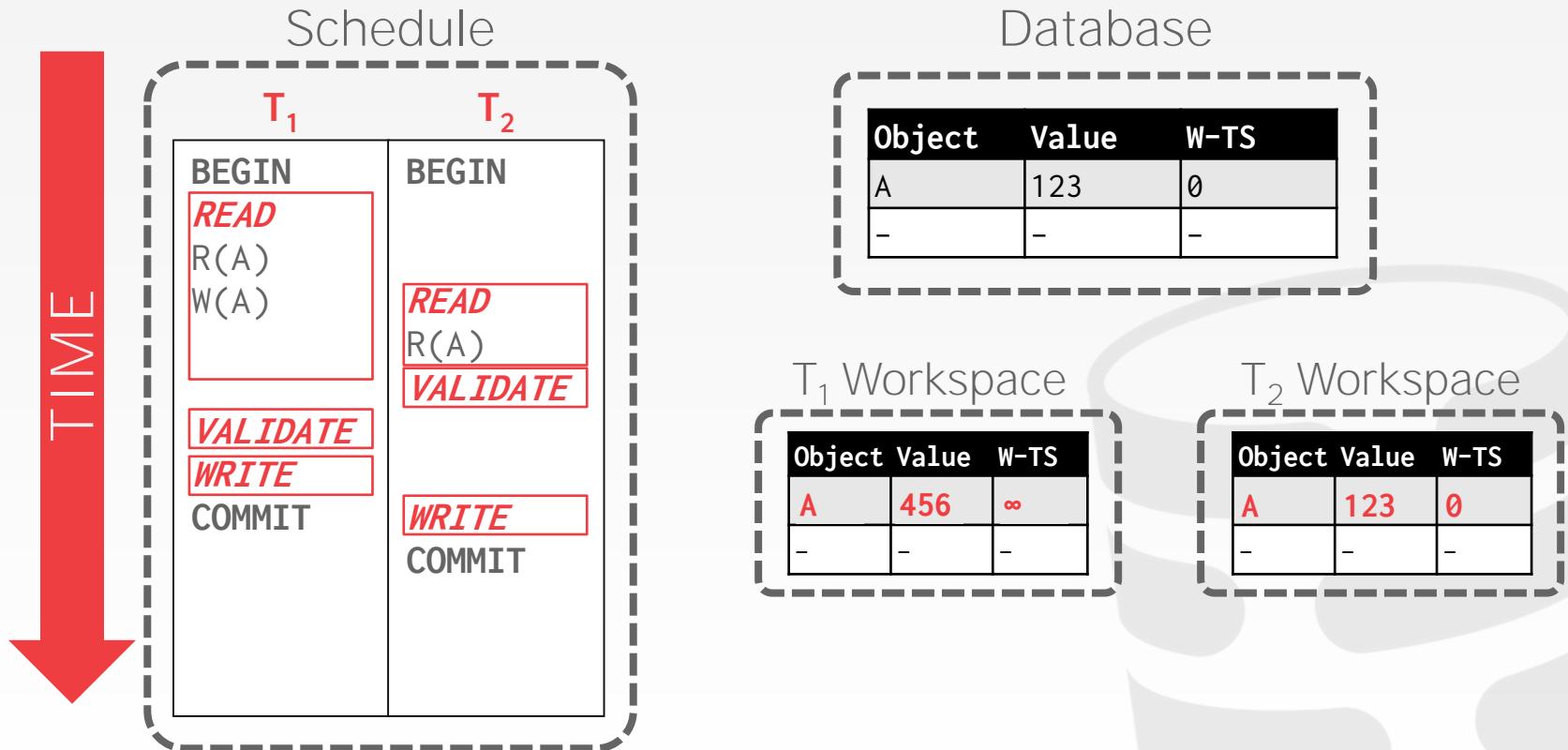
OCC – VALIDATION STEP #2



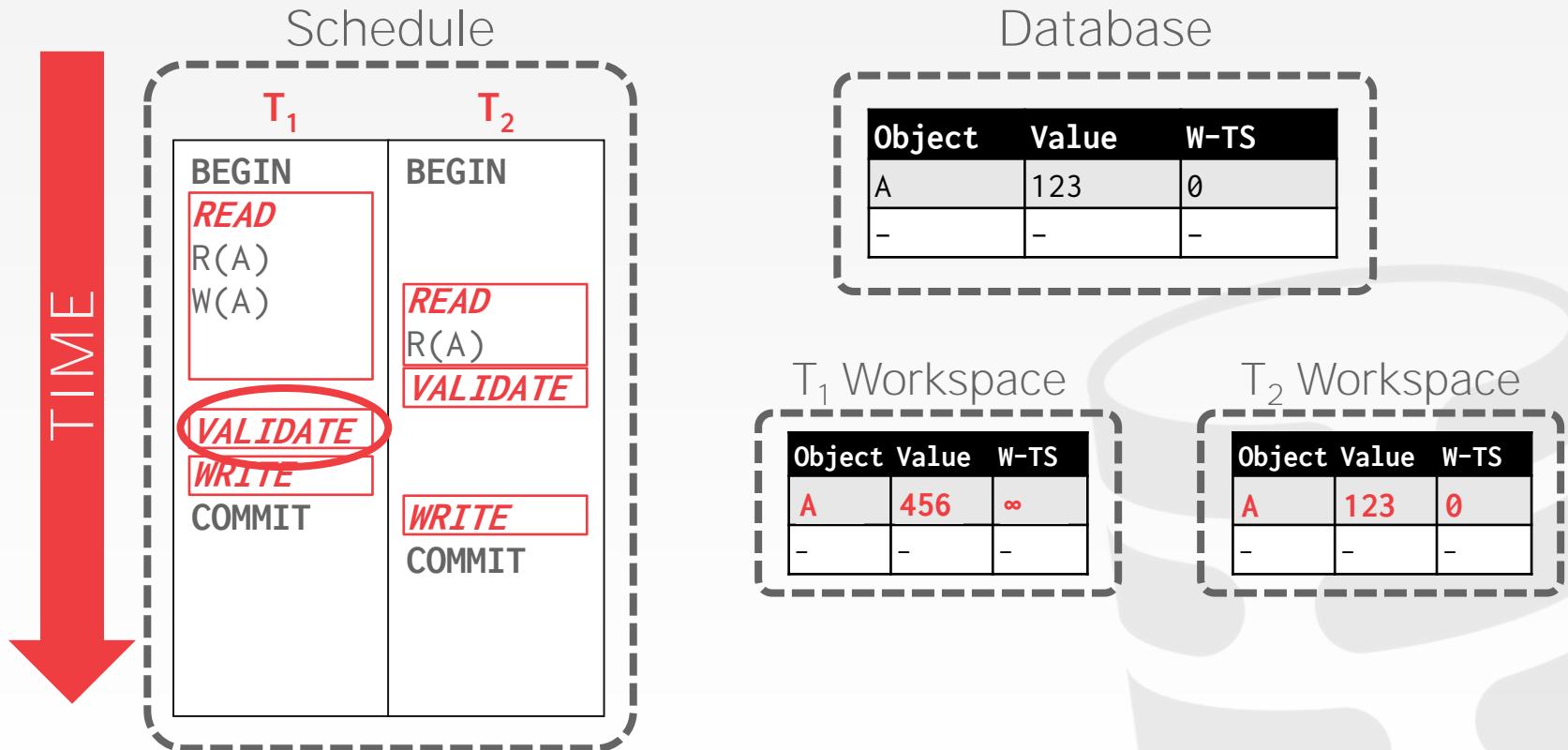
OCC – VALIDATION STEP #2



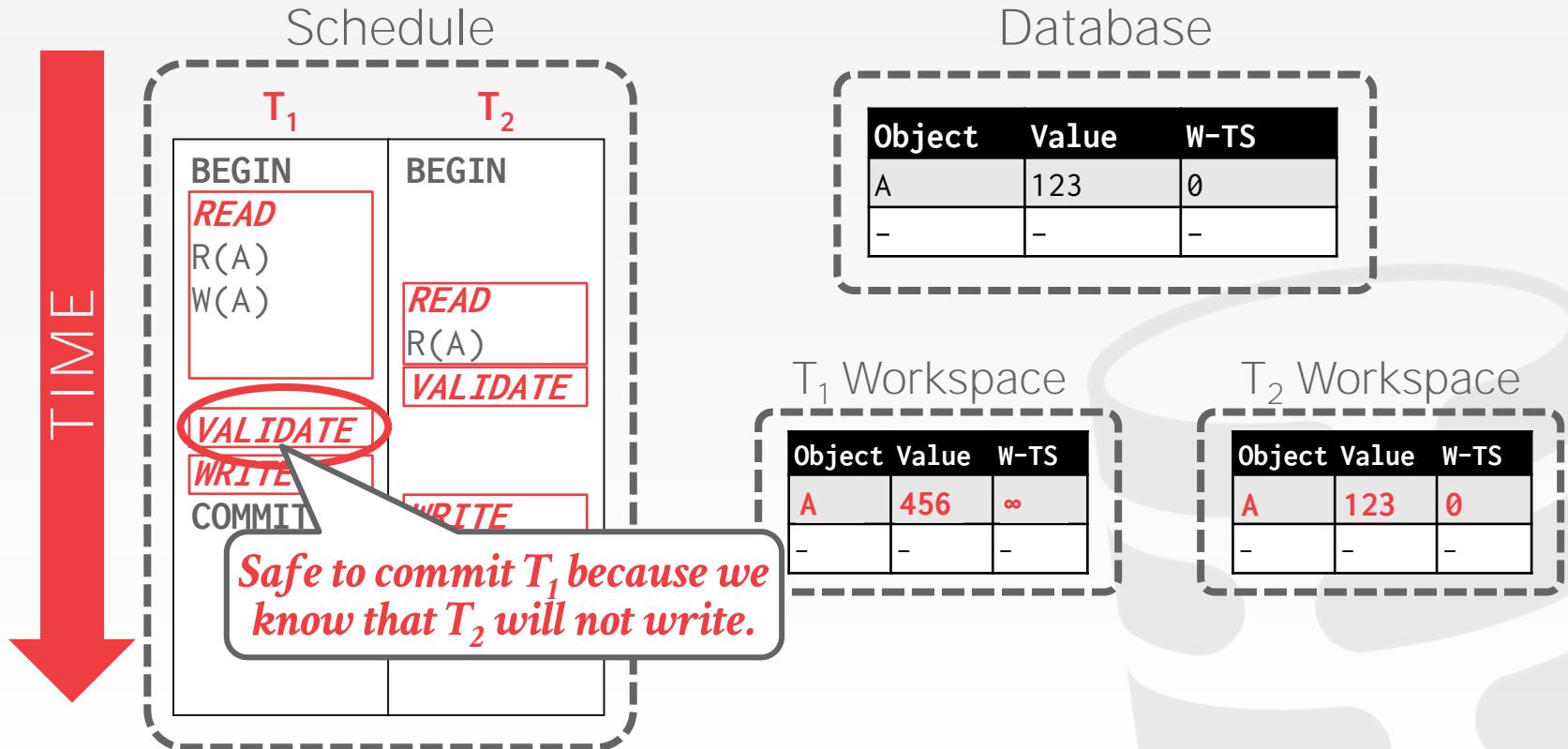
OCC – VALIDATION STEP #2



OCC – VALIDATION STEP #2



OCC – VALIDATION STEP #2



OCC – VALIDATION STEP #3

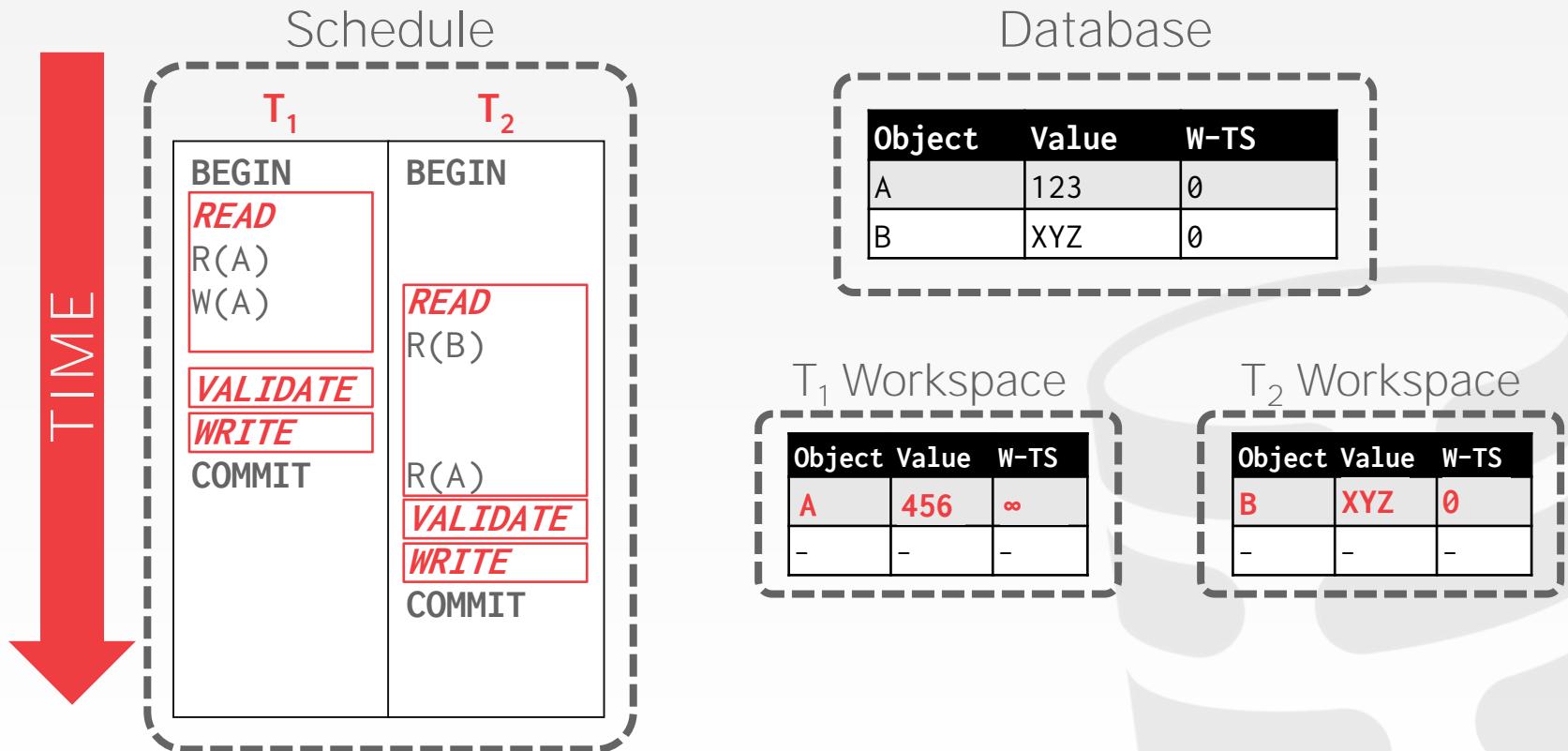
T_i completes its **Read** phase before T_j completes its **Read** phase

And T_i does not write to any object that is either read or written by T_j :

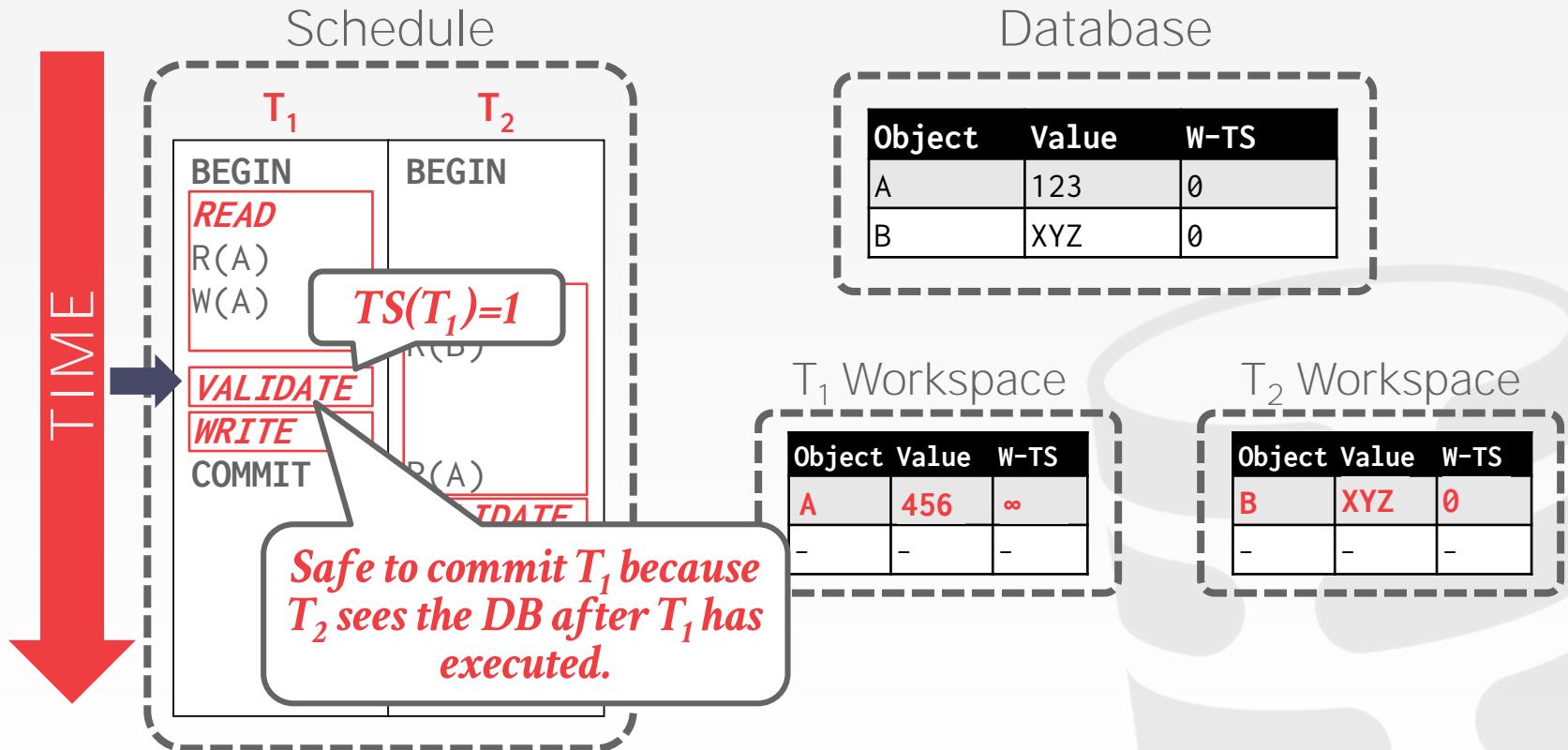
- $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j) = \emptyset$
- $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j) = \emptyset$



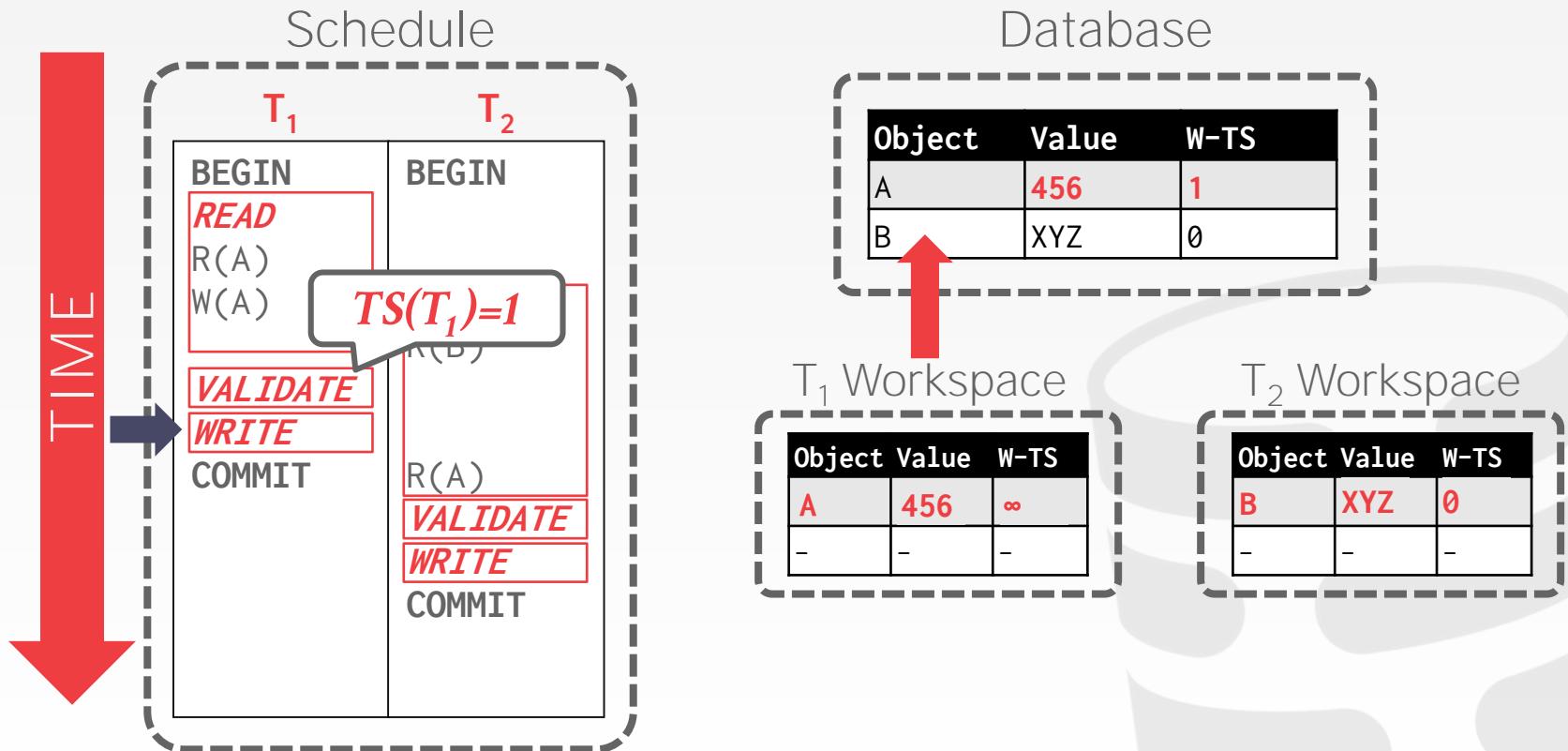
OCC – VALIDATION STEP #3



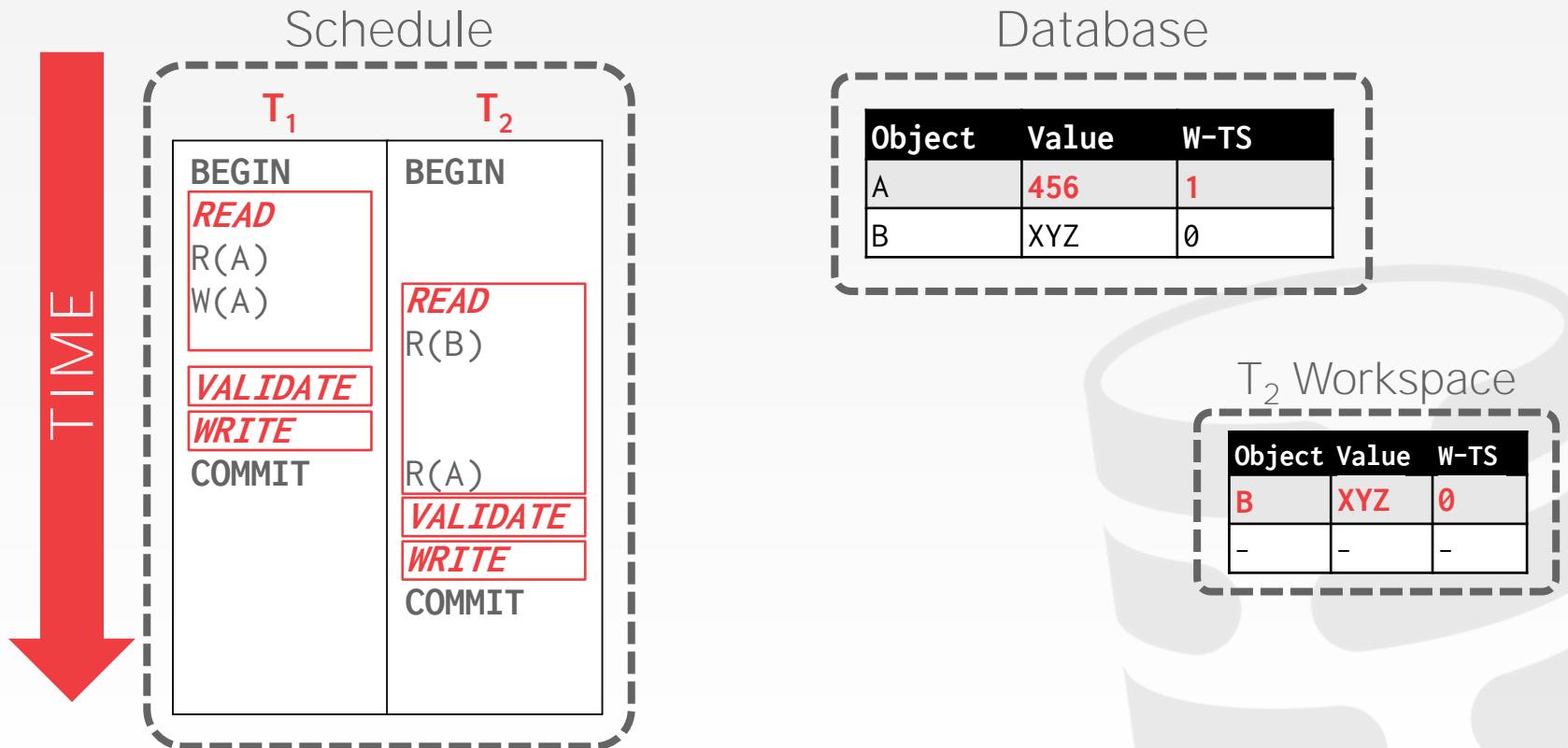
OCC – VALIDATION STEP #3



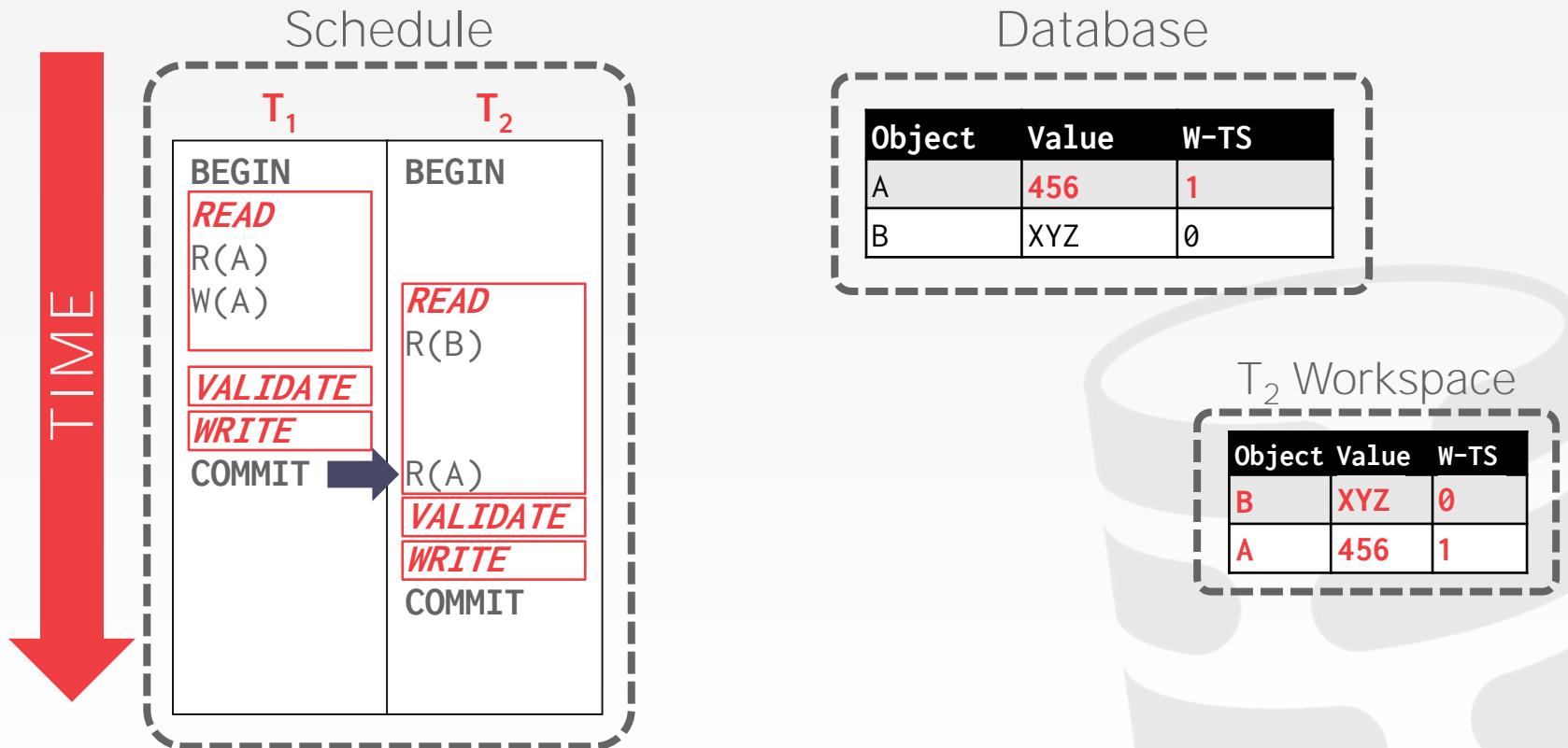
OCC – VALIDATION STEP #3



OCC – VALIDATION STEP #3



OCC – VALIDATION STEP #3

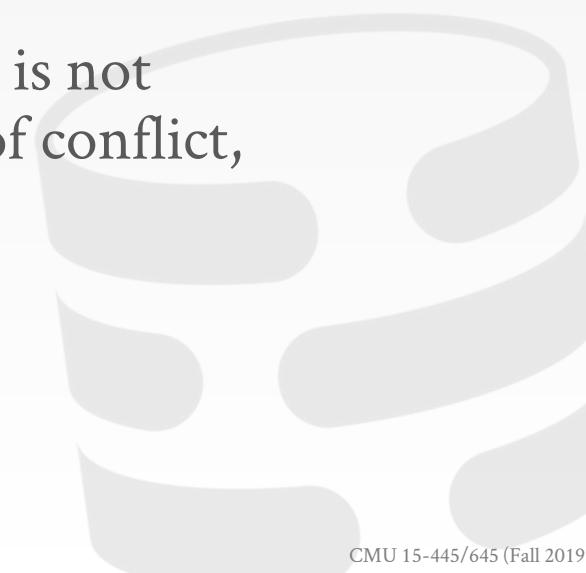


OCC – OBSERVATIONS

OCC works well when the # of conflicts is low:

- All txns are read-only (ideal).
- Txns access disjoint subsets of data.

If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

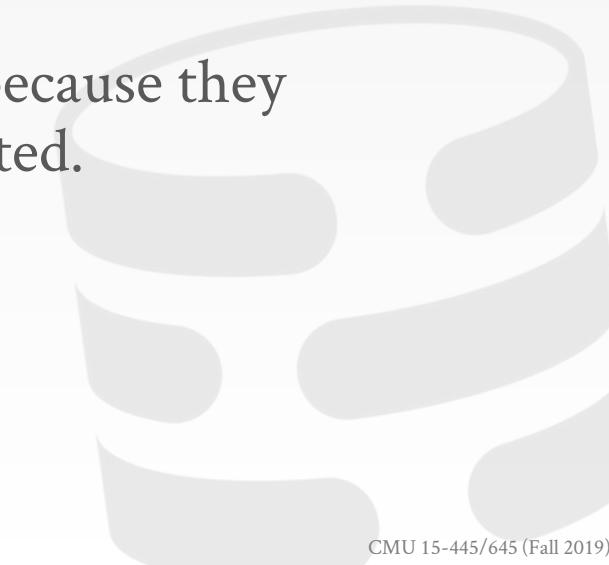


OCC – PERFORMANCE ISSUES

High overhead for copying data locally.

Validation/Write phase bottlenecks.

Aborts are more wasteful than in 2PL because they only occur after a txn has already executed.



OBSERVATION

When a txn commits, all previous T/O schemes check to see whether there is a conflict with concurrent txns.

→ This requires latches.

If you have a lot of concurrent txns, then this is slow even if the conflict rate is low.



PARTITION-BASED T/O

Split the database up in disjoint subsets called ***horizontal partitions*** (aka shards).

Use timestamps to order txns for serial execution at each partition.

→ Only check for conflicts between txns that are running in the same partition.

DATABASE PARTITIONING

```
CREATE TABLE customer (
    c_id INT PRIMARY KEY,
    c_email VARCHAR UNIQUE,
    ...
);
```

```
CREATE TABLE orders (
    o_id INT PRIMARY KEY,
    o_c_id INT REFERENCES
        ↳customer (c_id),
    ...
);
```

```
CREATE TABLE oitems (
    oi_id INT PRIMARY KEY,
    oi_o_id INT REFERENCES
        ↳orders (o_id),
    oi_c_id INT REFERENCES
        ↳orders (o_c_id),
    ...
);
```

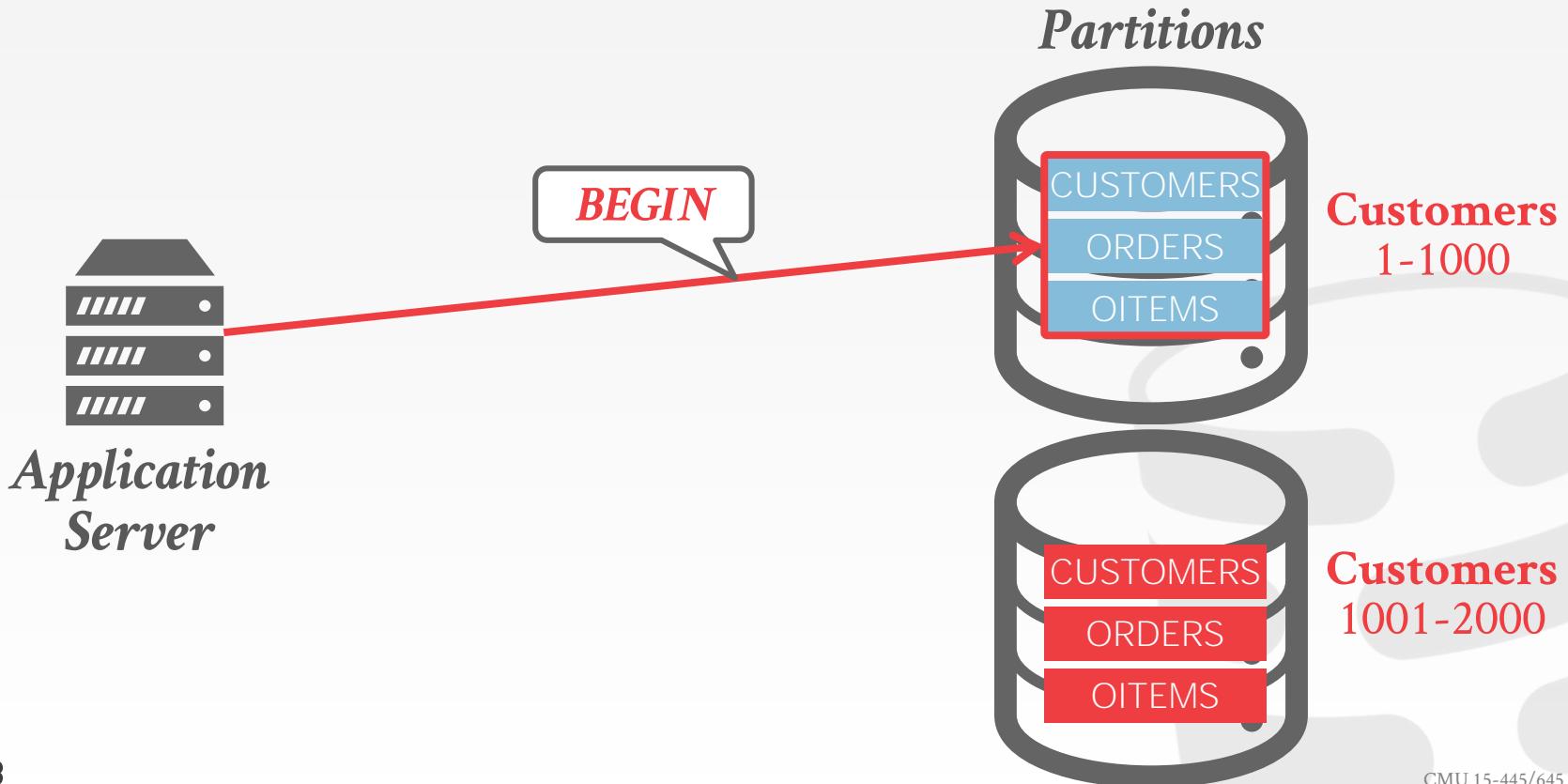
DATABASE PARTITIONING

```
CREATE TABLE customer (
    c_id INT PRIMARY KEY,
    c_email VARCHAR UNIQUE,
    ...
);
```

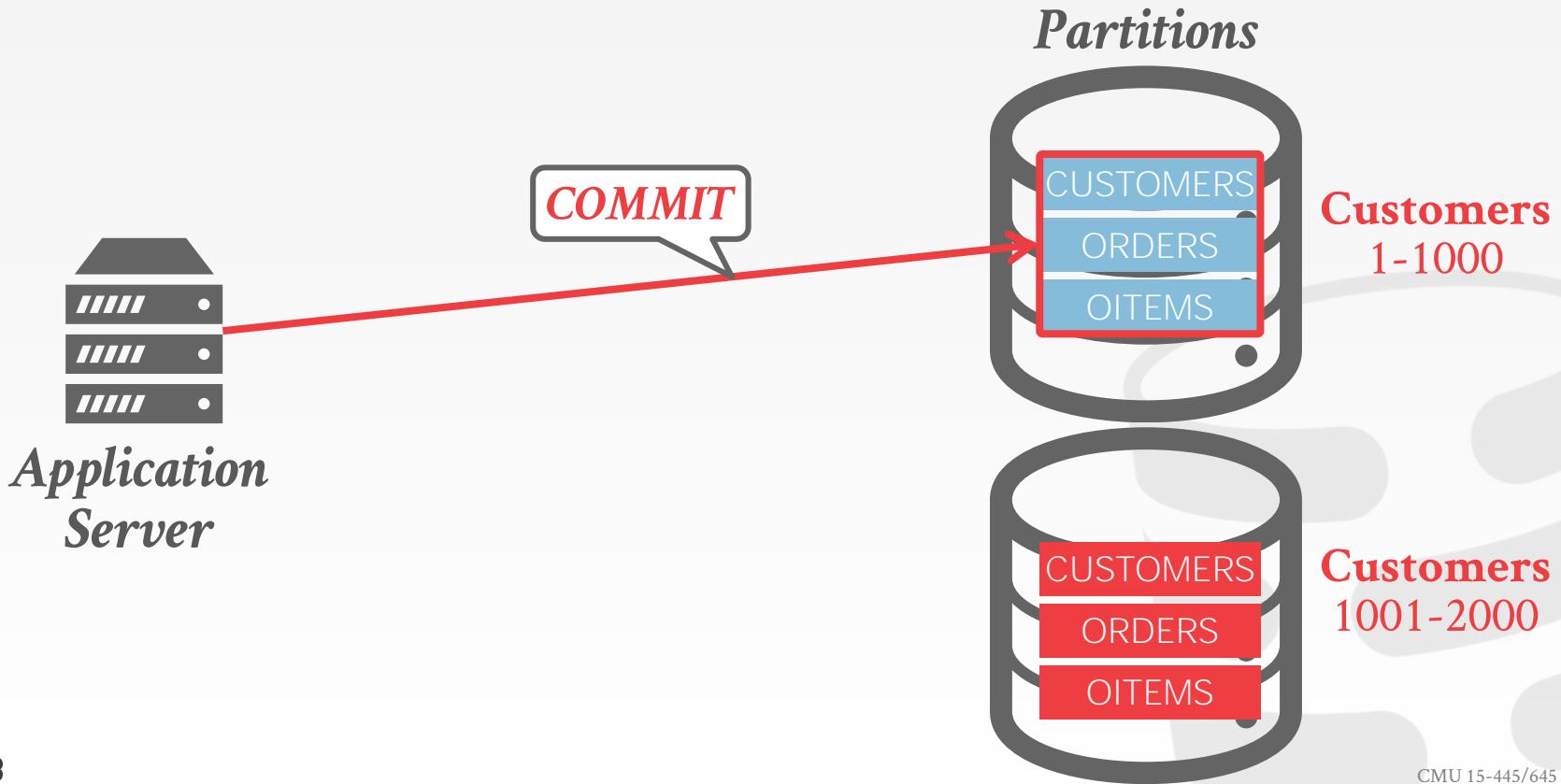
```
CREATE TABLE orders (
    o_id INT PRIMARY KEY,
    o_c_id INT REFERENCES
        ↳ customer (c_id),
    ...
);
```

```
CREATE TABLE oitems (
    oi_id INT PRIMARY KEY,
    oi_o_id INT REFERENCES
        ↳ orders (o_id),
    oi_c_id INT REFERENCES
        ↳ orders (o_c_id),
    ...
);
```

HORIZONTAL PARTITIONING



HORIZONTAL PARTITIONING



PARTITION-BASED T/O

Txns are assigned timestamps based on when they arrive at the DBMS.

Partitions are protected by a single lock:

- Each txn is queued at the partitions it needs.
- The txn acquires a partition's lock if it has the lowest timestamp in that partition's queue.
- The txn starts when it has all of the locks for all the partitions that it will read/write.



PARTITION-BASED T/O – READS

Txns can read anything that they want at the partitions that they have locked.

If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.



PARTITION-BASED T/O – WRITES

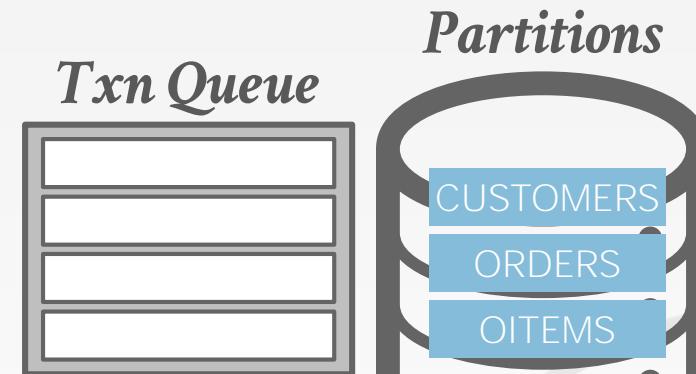
All updates occur in place.

→ Maintain a separate in-memory buffer to undo changes if the txn aborts.

If a txn tries to write to a partition that it does not have the lock, it is aborted + restarted.



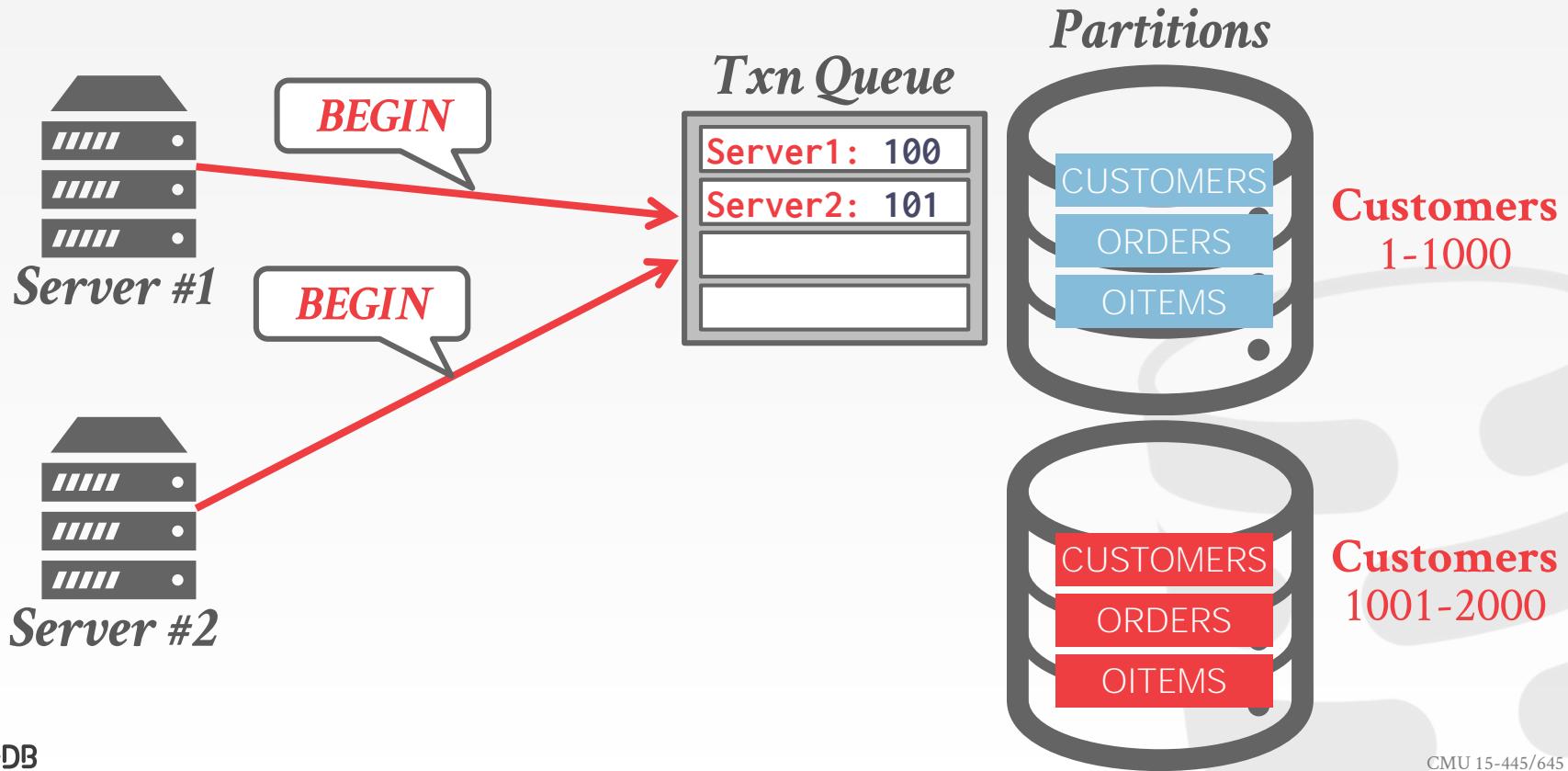
PARTITION-BASED T/O



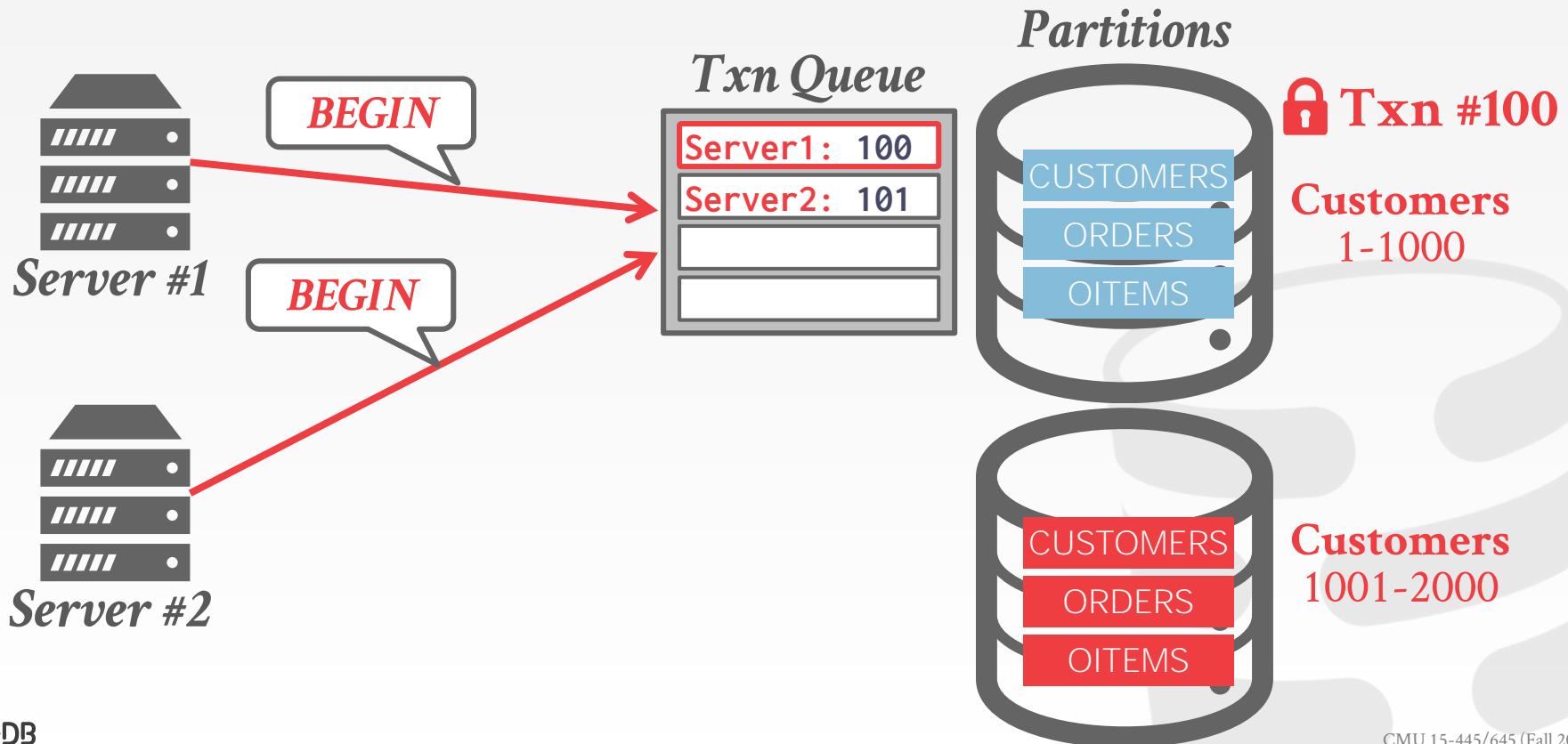
Customers
1-1000

Customers
1001-2000

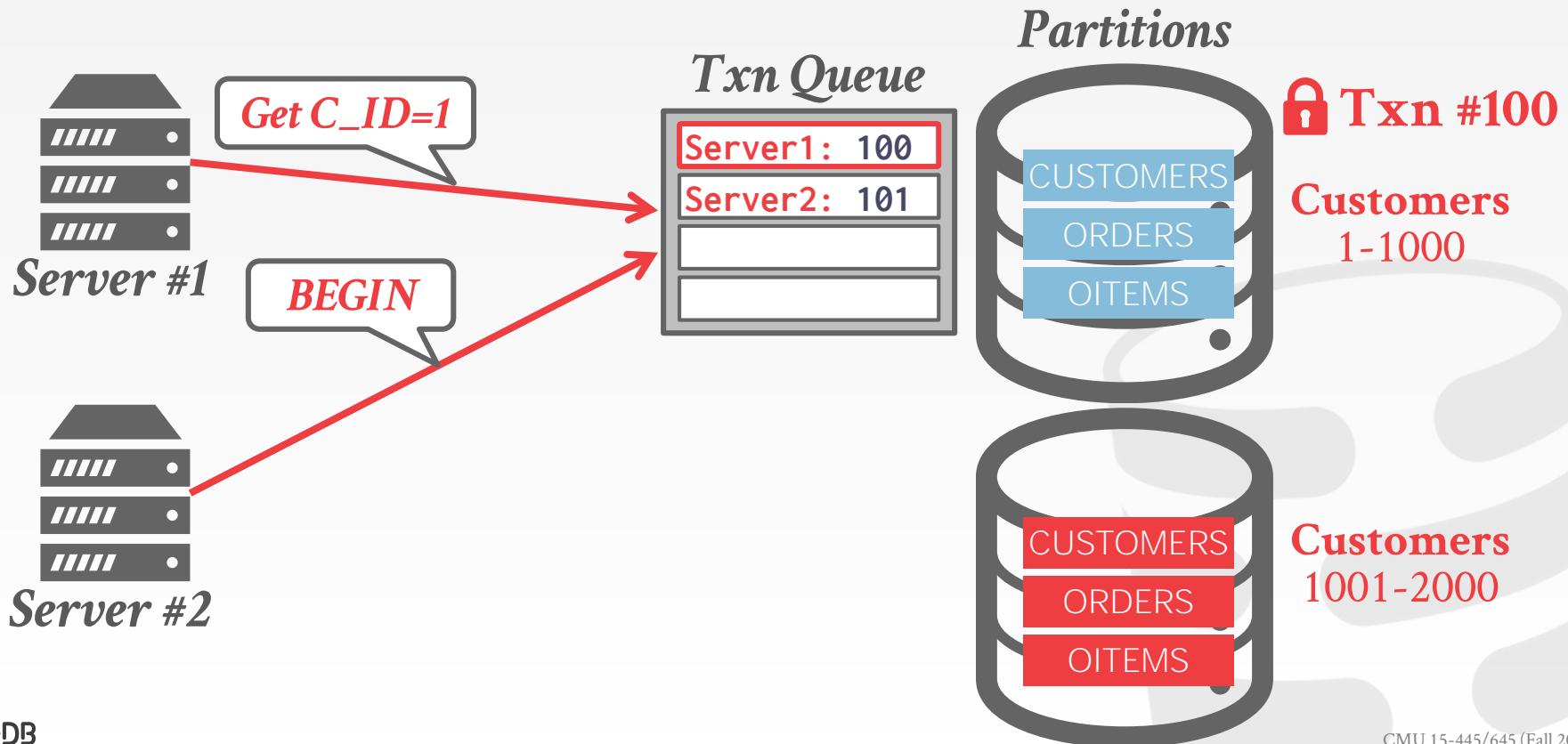
PARTITION-BASED T/O



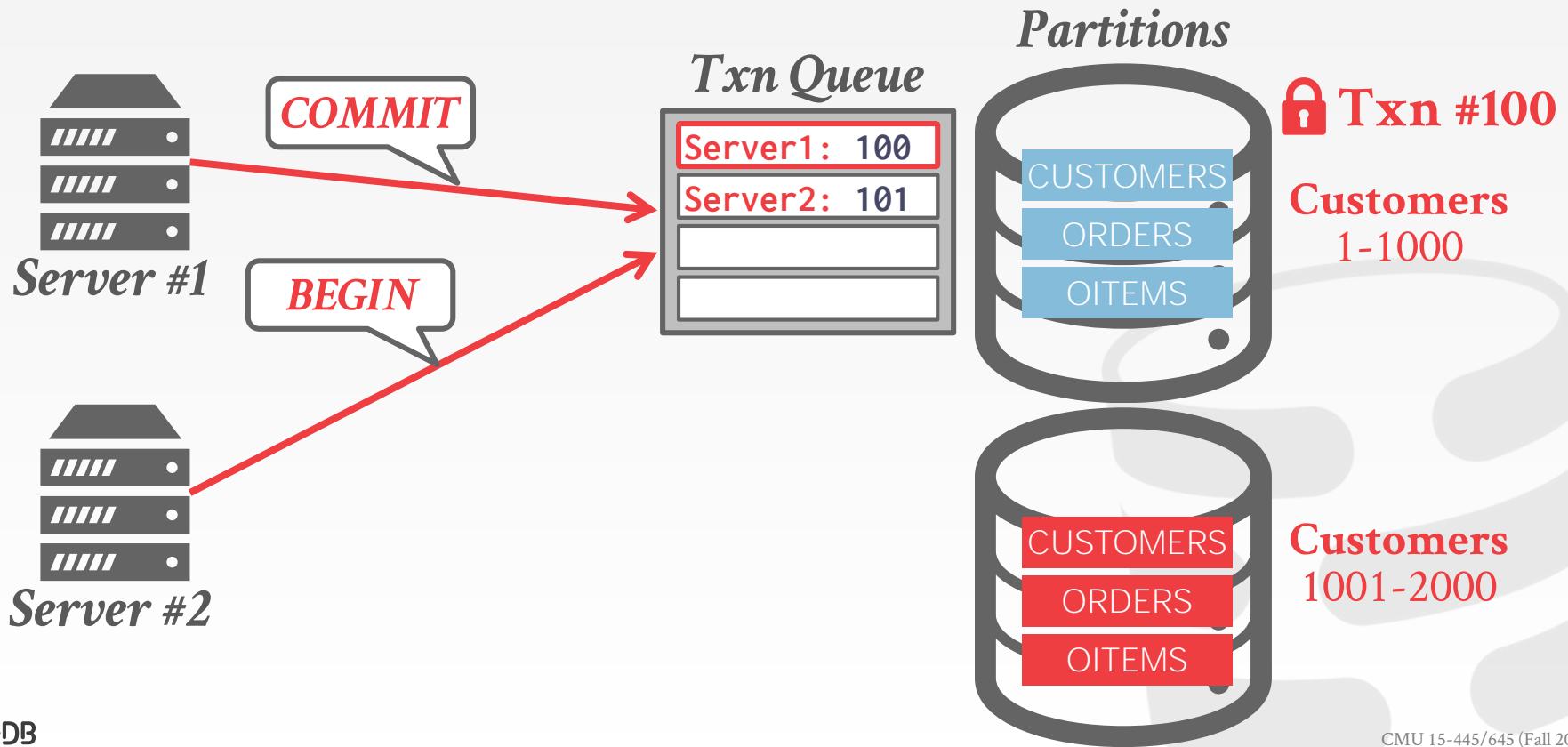
PARTITION-BASED T/O



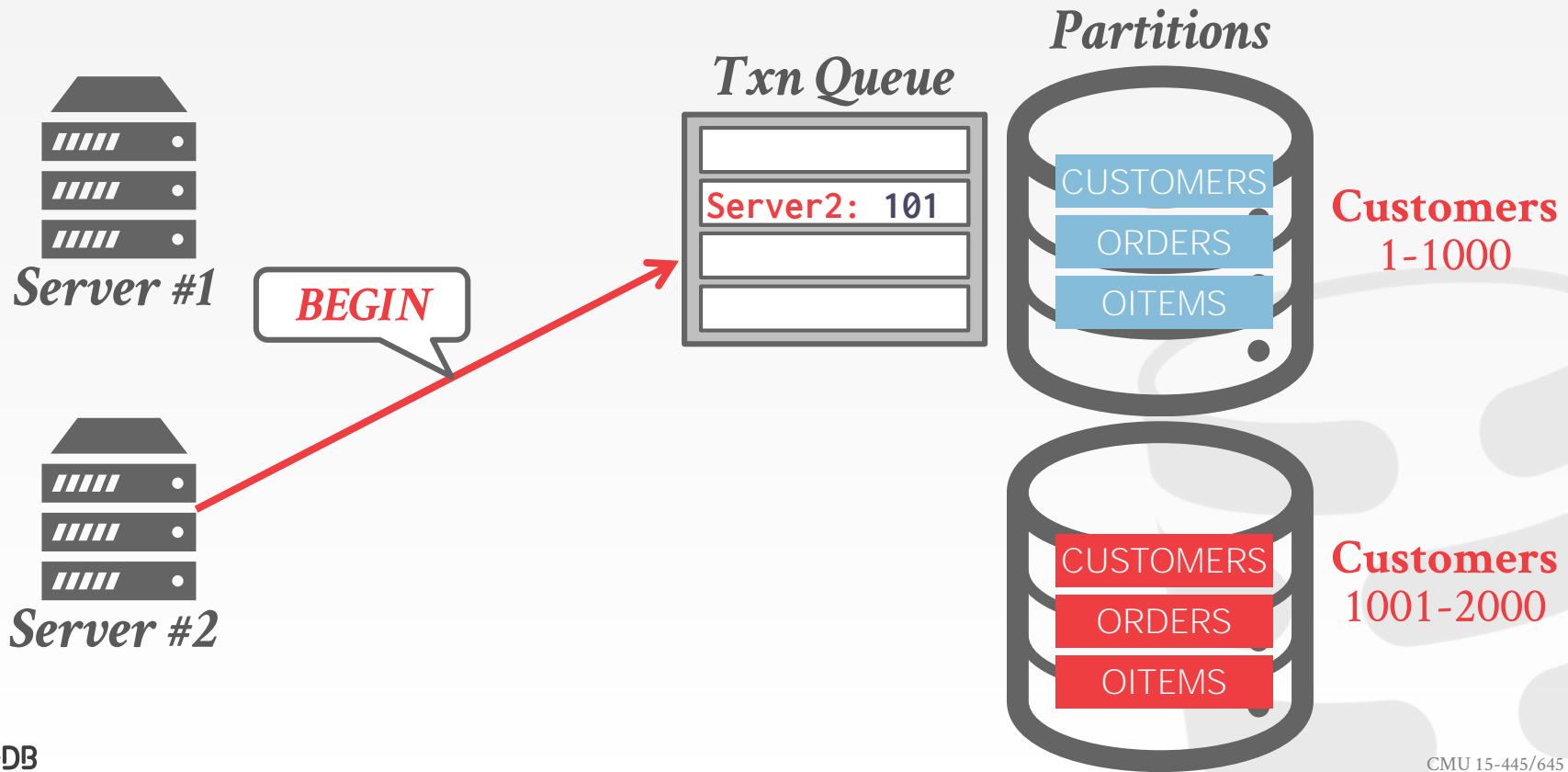
PARTITION-BASED T/O



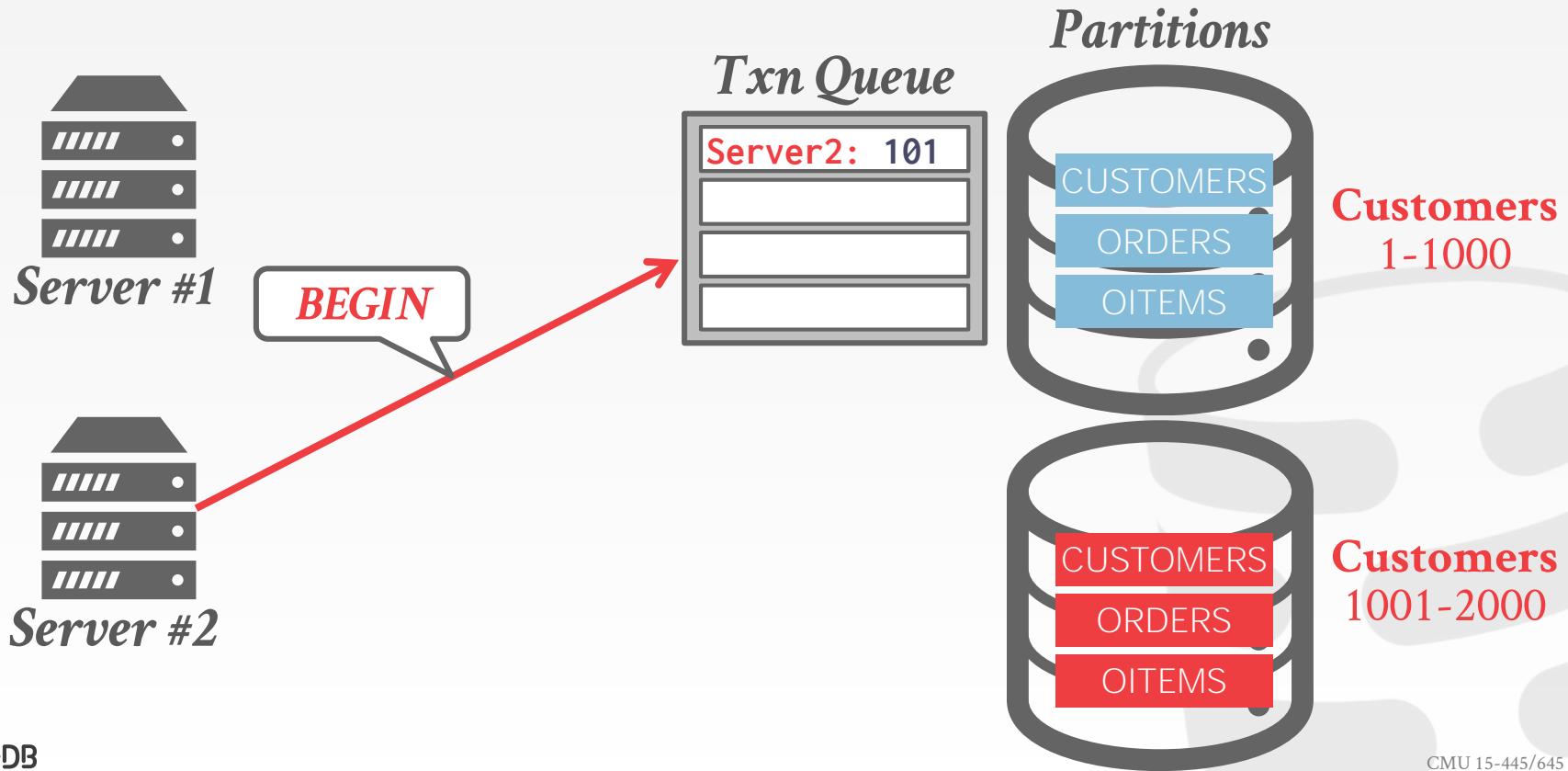
PARTITION-BASED T/O



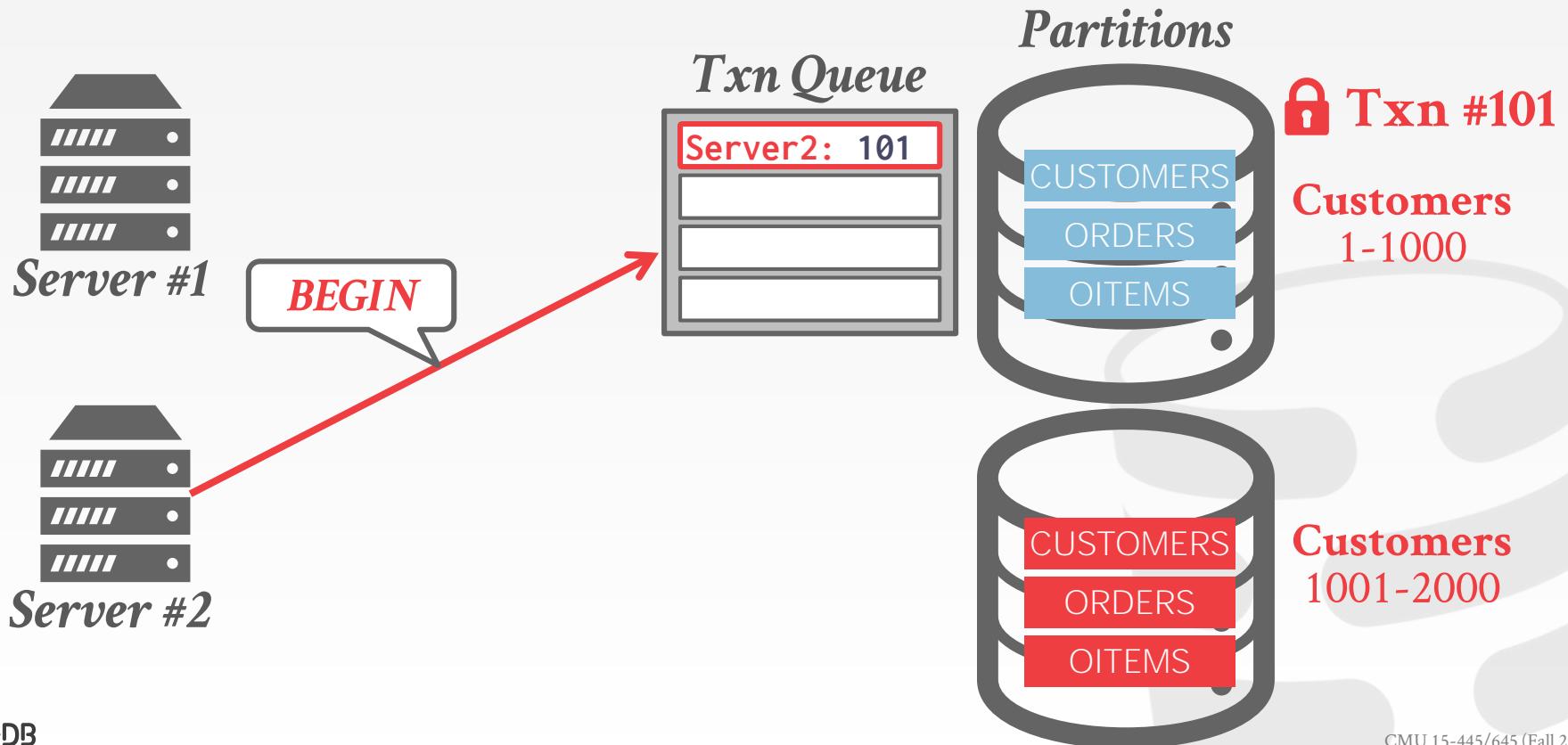
PARTITION-BASED T/O



PARTITION-BASED T/O



PARTITION-BASED T/O



PARTITIONED T/O – PERFORMANCE ISSUES

Partition-based T/O protocol is fast if:

- The DBMS knows what partitions the txn needs before it starts.
- Most (if not all) txns only need to access a single partition.

Multi-partition txns causes partitions to be idle while txn executes.



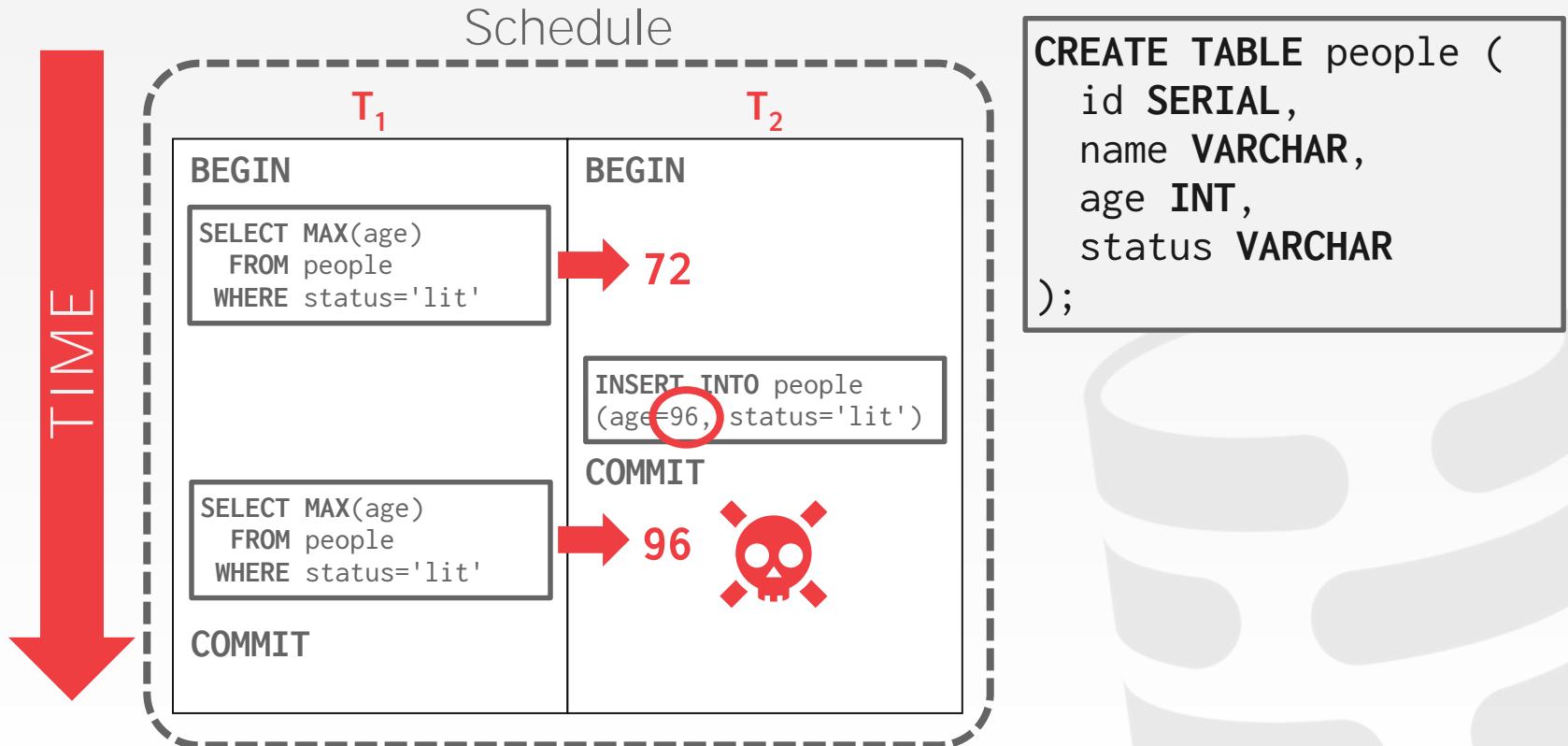
DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

But now if we have insertions, updates, and deletions, we have new problems...



THE PHANTOM PROBLEM



WTF?

How did this happen?

- Because T_1 locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.

PREDICATE LOCKING

Lock records that satisfy a logical predicate:

→ Example: **status='lit'**

In general, predicate locking has a lot of locking overhead.

Index locking is a special case of predicate locking that is potentially more efficient.



INDEX LOCKING

If there is a dense index on the status field then the txn can lock index page containing the data with **status='lit'**.

If there are no records with **status='lit'**, the txn must lock the index page where such a data entry would be, if it existed.

LOCKING WITHOUT AN INDEX

If there is no suitable index, then the txn must obtain:

- A lock on every page in the table to prevent a record's **status='lit'** from being changed to **lit**.
- The lock for the table itself to prevent records with **status='lit'** from being added or deleted.



REPEATING SCANS

An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.

- Have to retain the scan set for every range query in a txn.
- Andy doesn't know of any commercial system that does this (only just Silo?).

WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

We may want to use a weaker level of consistency to improve scalability.



ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:

- Dirty Reads
- Unrepeatable Reads
- Phantom Reads



ISOLATION LEVELS

Isolation (High → Low)

SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS: Phantoms may happen.

READ COMMITTED: Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED: All of them may happen.

ISOLATION LEVELS

	Dirty Read	Unrepeatable Read	Phantom
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe

ISOLATION LEVELS

SERIALIZABLE: Obtain all locks first; plus index locks, plus strict 2PL.

REPEATABLE READS: Same as above, but no index locks.

READ COMMITTED: Same as above, but **S** locks are released immediately.

READ UNCOMMITTED: Same as above, but allows dirty reads (no **S** locks).

SQL-92 ISOLATION LEVELS

You set a txn's isolation level before you execute any queries in that txn.

Not all DBMS support all isolation levels in all execution scenarios

→ Replicated Environments

The default depends on implementation...

```
SET TRANSACTION ISOLATION LEVEL  
<isolation-level>;
```

```
BEGIN TRANSACTION ISOLATION LEVEL  
<isolation-level>;
```

ISOLATION LEVELS (2013)

	Default	Maximum
Actian Ingres 10.0/10s	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: [Peter Bailis](#)

SQL-92 ACCESS MODES

You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.

Only two possible modes:

- **READ WRITE** (Default)
- **READ ONLY**

Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

```
SET TRANSACTION <access-mode>;
```

```
BEGIN TRANSACTION <access-mode>;
```



CONCLUSION

Every concurrency control can be broken down into the basic concepts that I've described in the last two lectures.

I'm not showing benchmark results because I don't want you to get the wrong idea.



NEXT CLASS

Multi-Version Concurrency Control



19 |

Multi-Version Concurrency Control



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #3 is due Sun Nov 17th @ 11:59pm.

Homework #4 was released last week.
It is due Wed Nov 13th @ 11:59pm.



MULTI-VERSION CONCURRENCY CONTROL

The DBMS maintains multiple **physical** versions of a single **logical** object in the database:

- When a txn writes to an object, the DBMS creates a new version of that object.
- When a txn reads an object, it reads the newest version that existed when the txn started.



MVCC HISTORY

Protocol was first proposed in 1978 MIT PhD dissertation.

First implementations was Rdb/VMS and InterBase at DEC in early 1980s.

- Both were by Jim Starkey, co-founder of NuoDB.
- DEC Rdb/VMS is now "Oracle Rdb"
- InterBase was open-sourced as Firebird.



MULTI-VERSION CONCURRENCY CONTROL

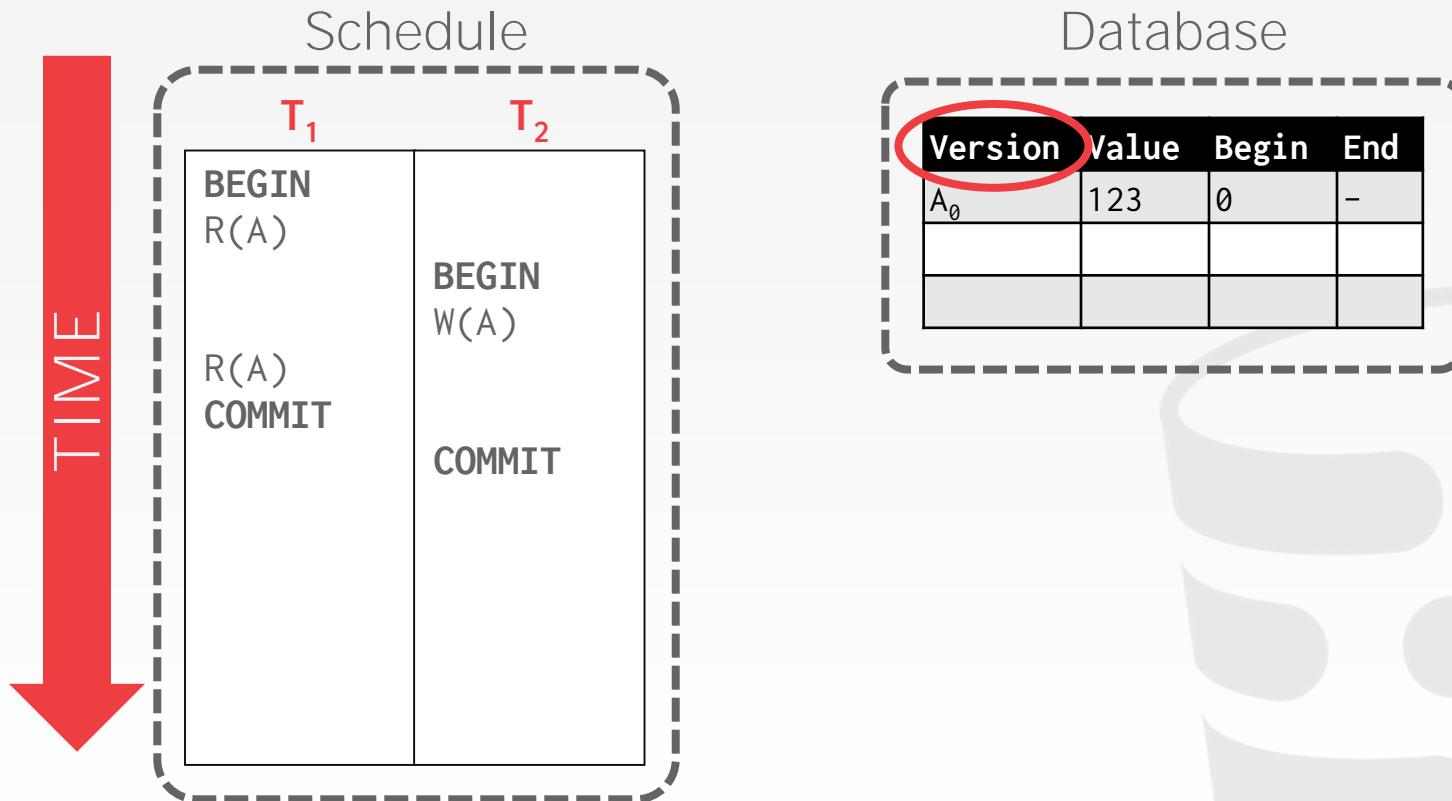
Writers don't block readers.
Readers don't block writers.

Read-only txns can read a consistent snapshot without acquiring locks.
→ Use timestamps to determine visibility.

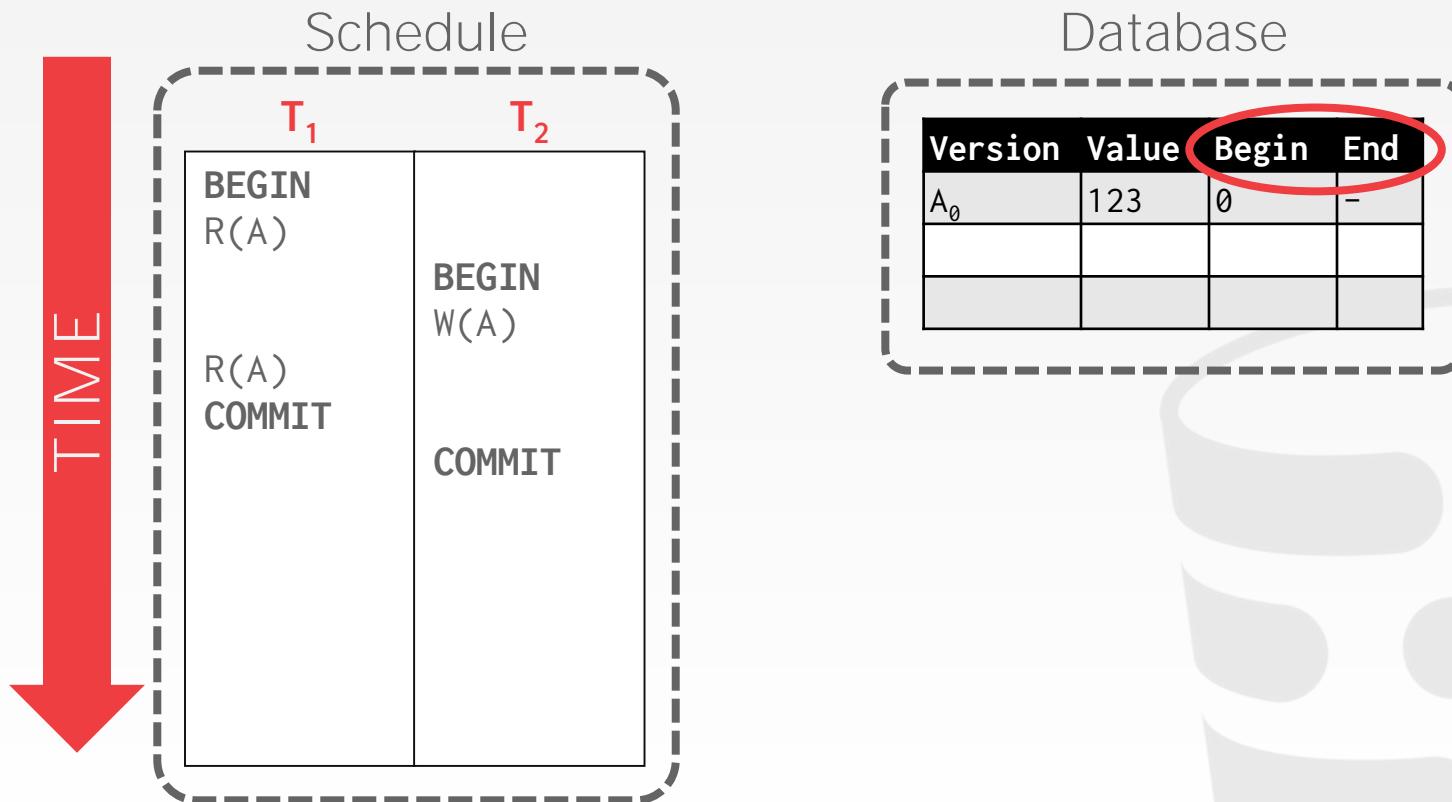
Easily support time-travel queries.



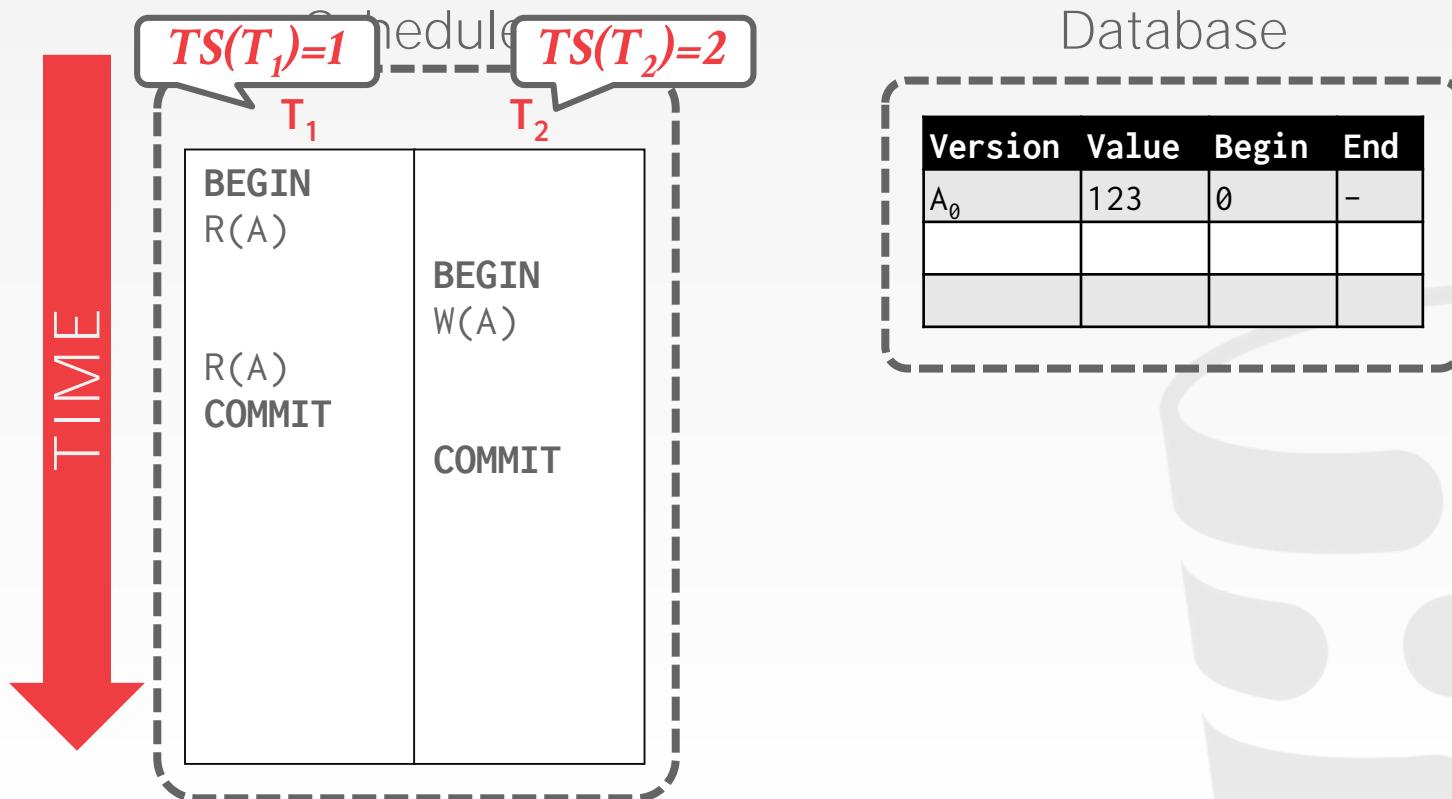
MVCC – EXAMPLE #1



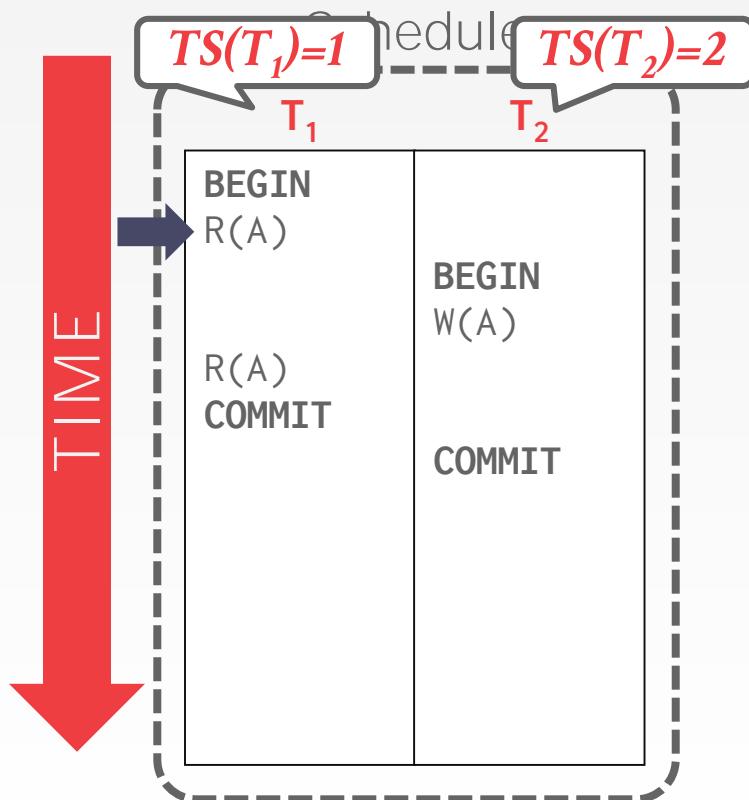
MVCC – EXAMPLE #1



MVCC – EXAMPLE #1



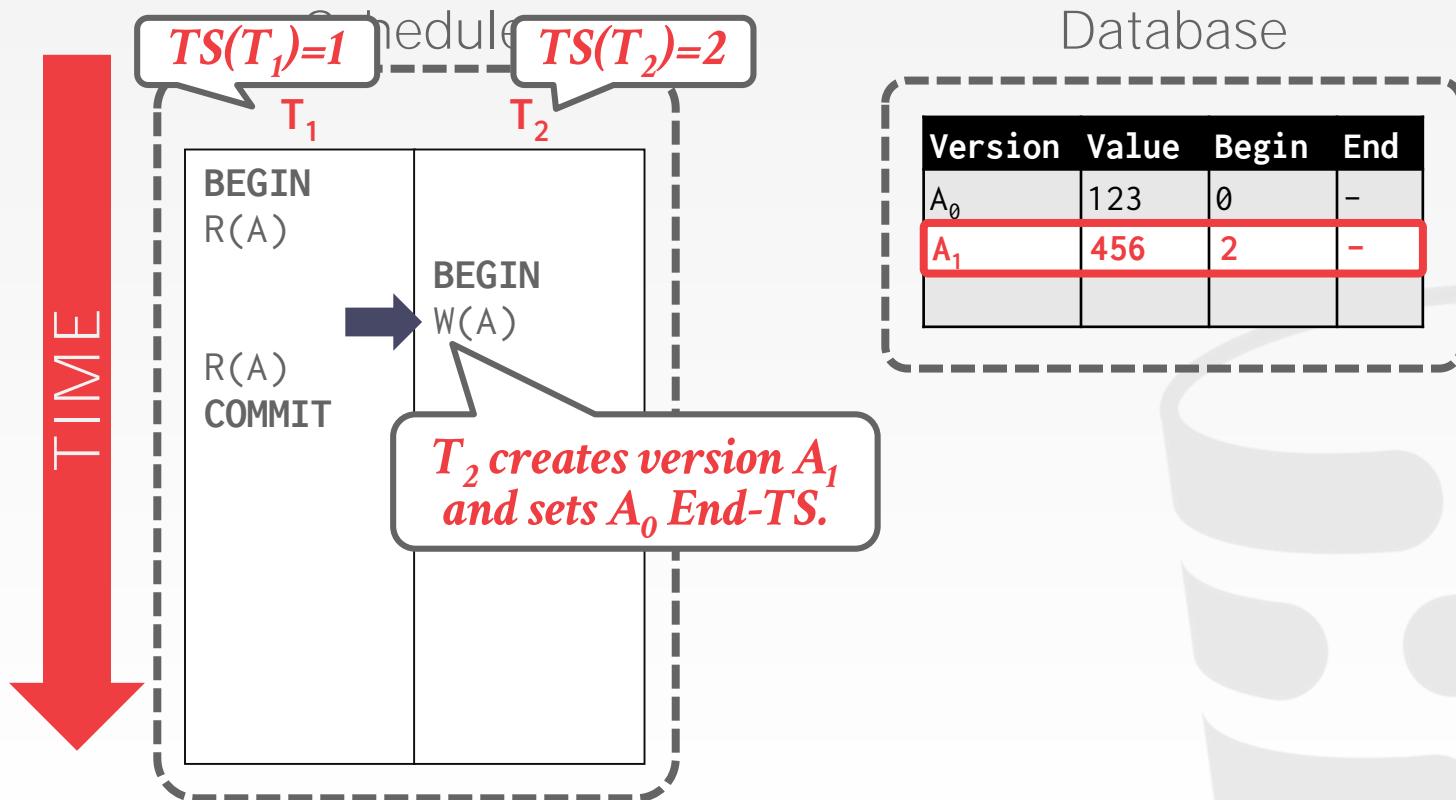
MVCC – EXAMPLE #1



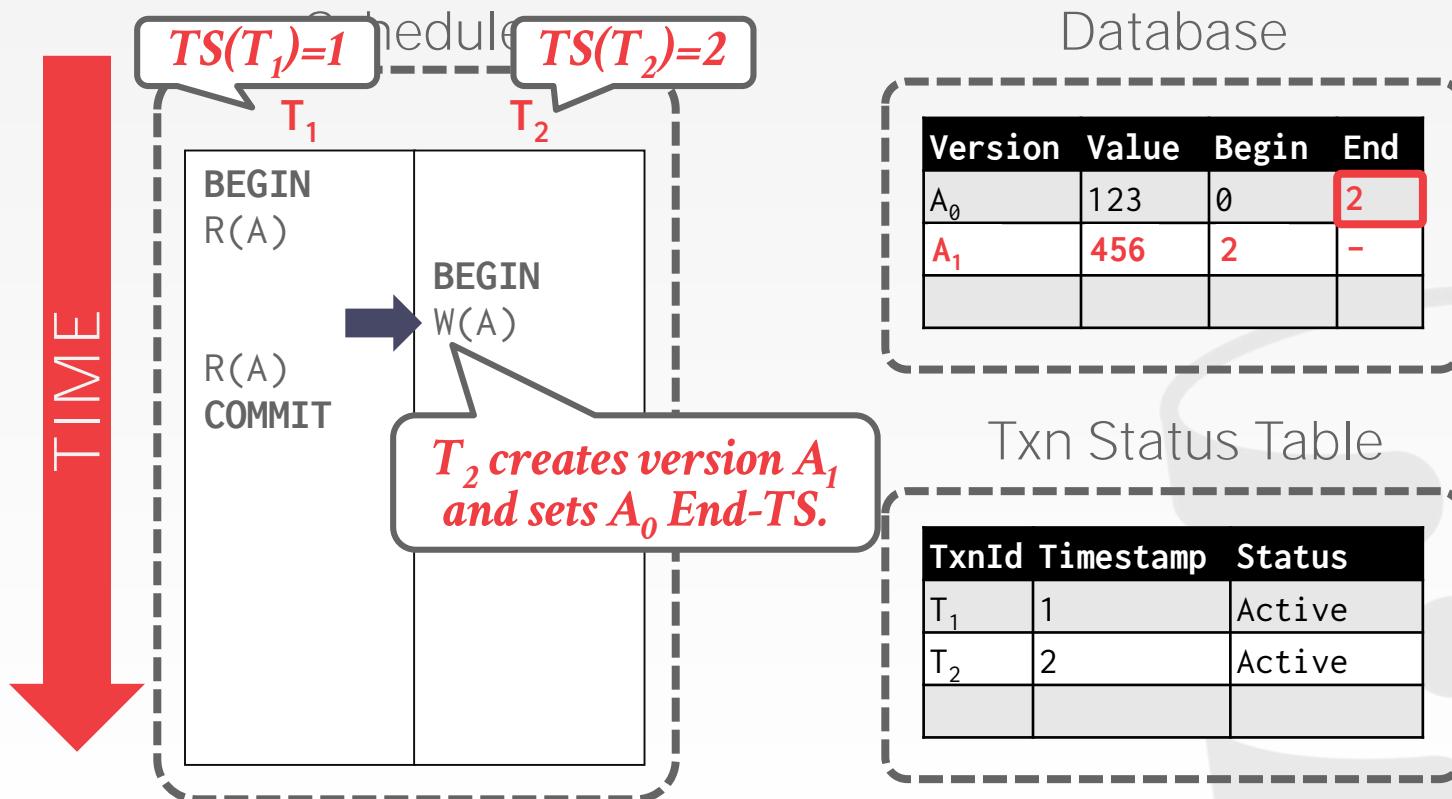
Database

Version	Value	Begin	End
A ₀	123	0	-

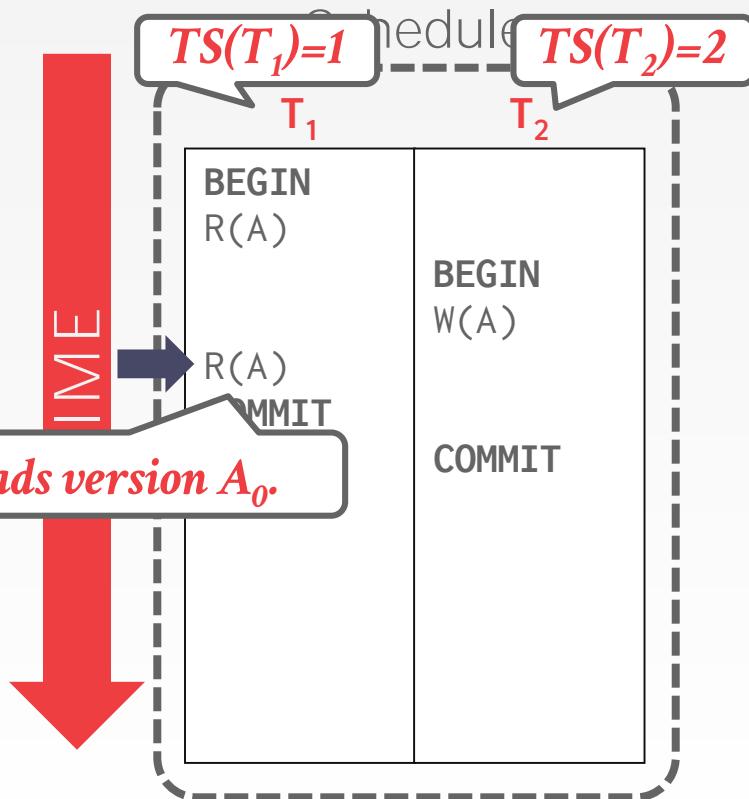
MVCC – EXAMPLE #1



MVCC – EXAMPLE #1



MVCC – EXAMPLE #1



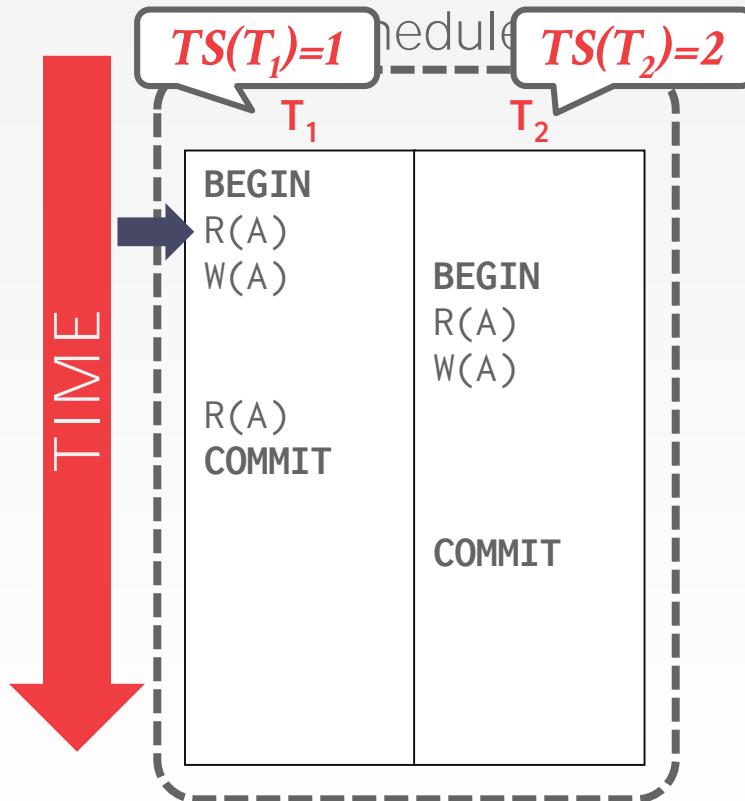
Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

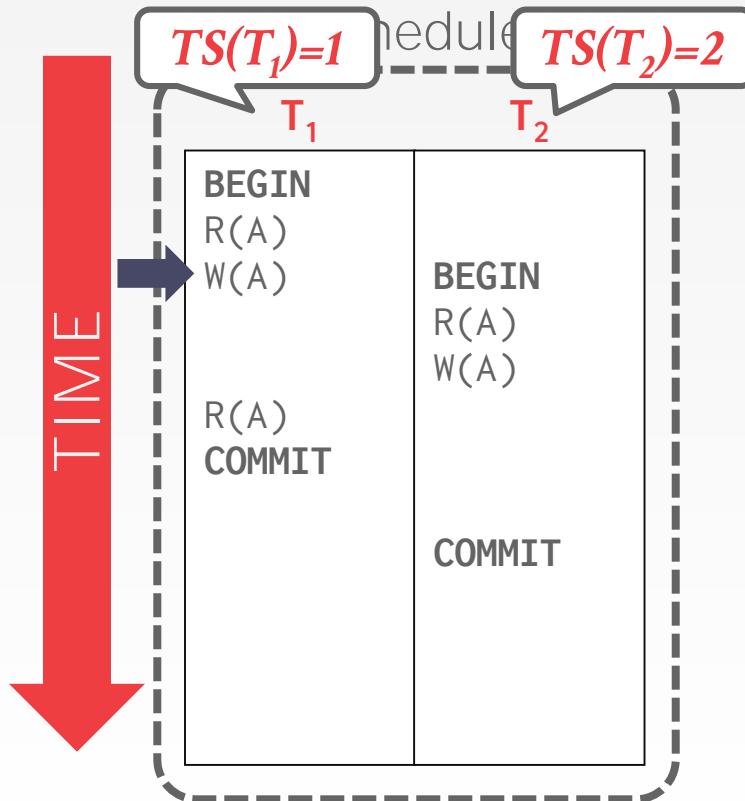
MVCC – EXAMPLE #2



Database			
Version	Value	Begin	End
A_0	123	0	

Txn Status Table		
TxnId	Timestamp	Status
T_1	1	Active

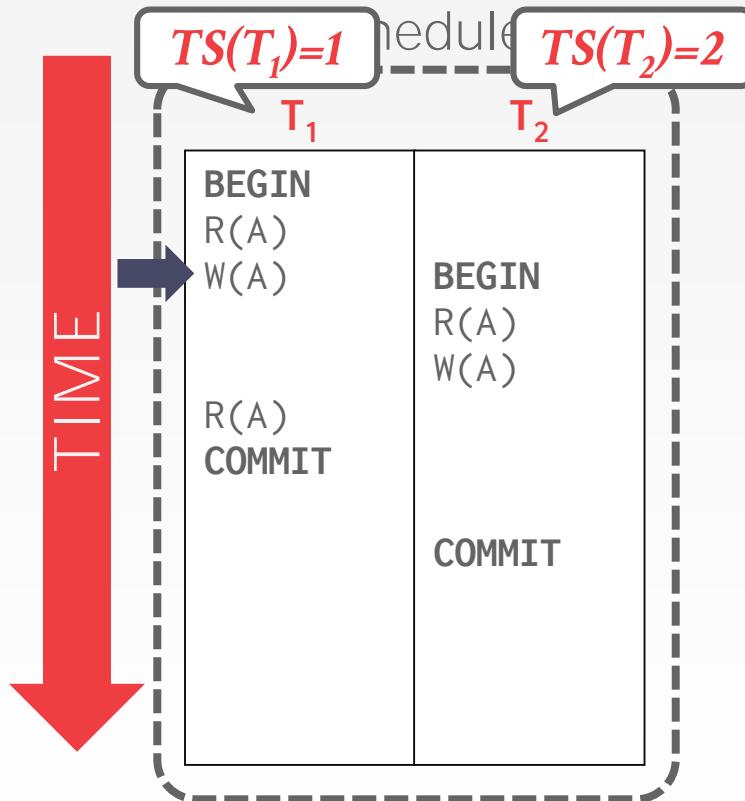
MVCC – EXAMPLE #2



Database			
Version	Value	Begin	End
A_0	123	0	

Txn Status Table		
TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



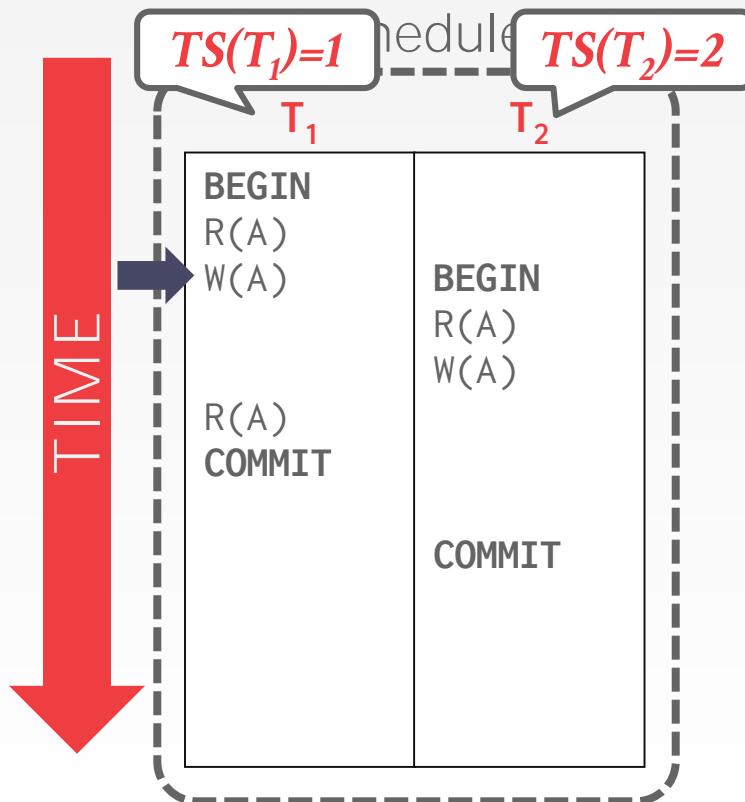
Database

Version	Value	Begin	End
A_0	123	0	
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



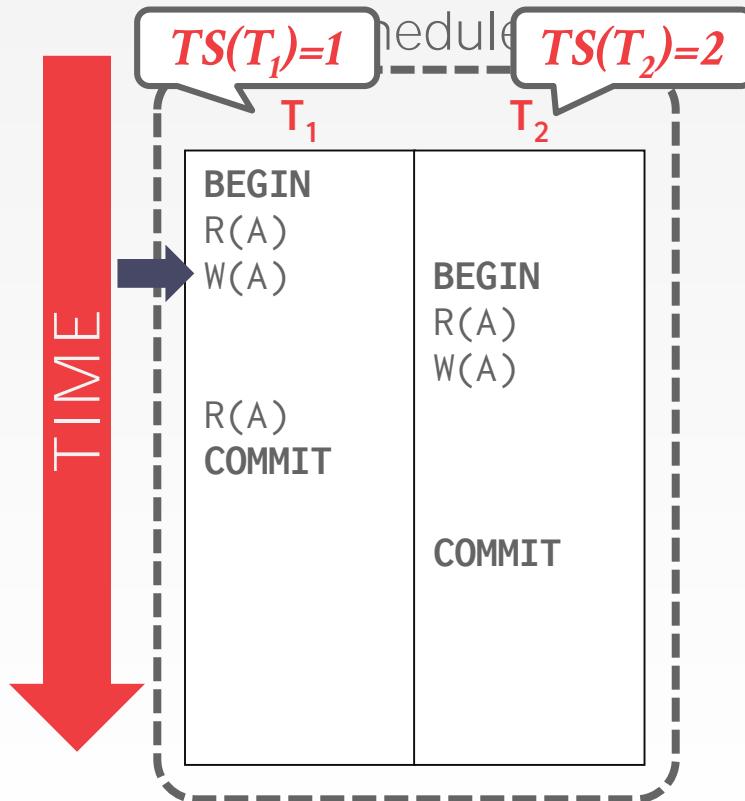
Database

Version	Value	Begin	End
A_0	123	0	
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



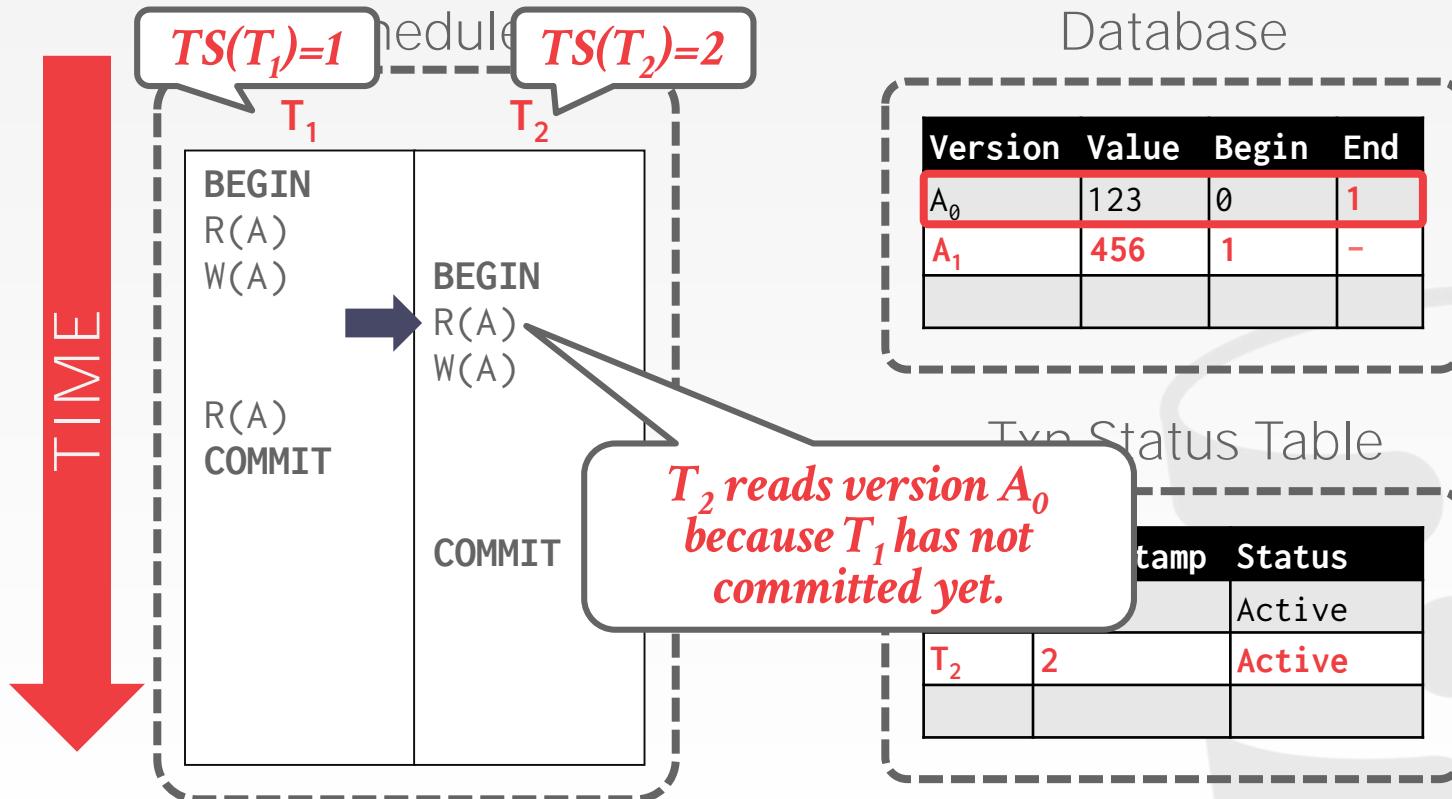
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

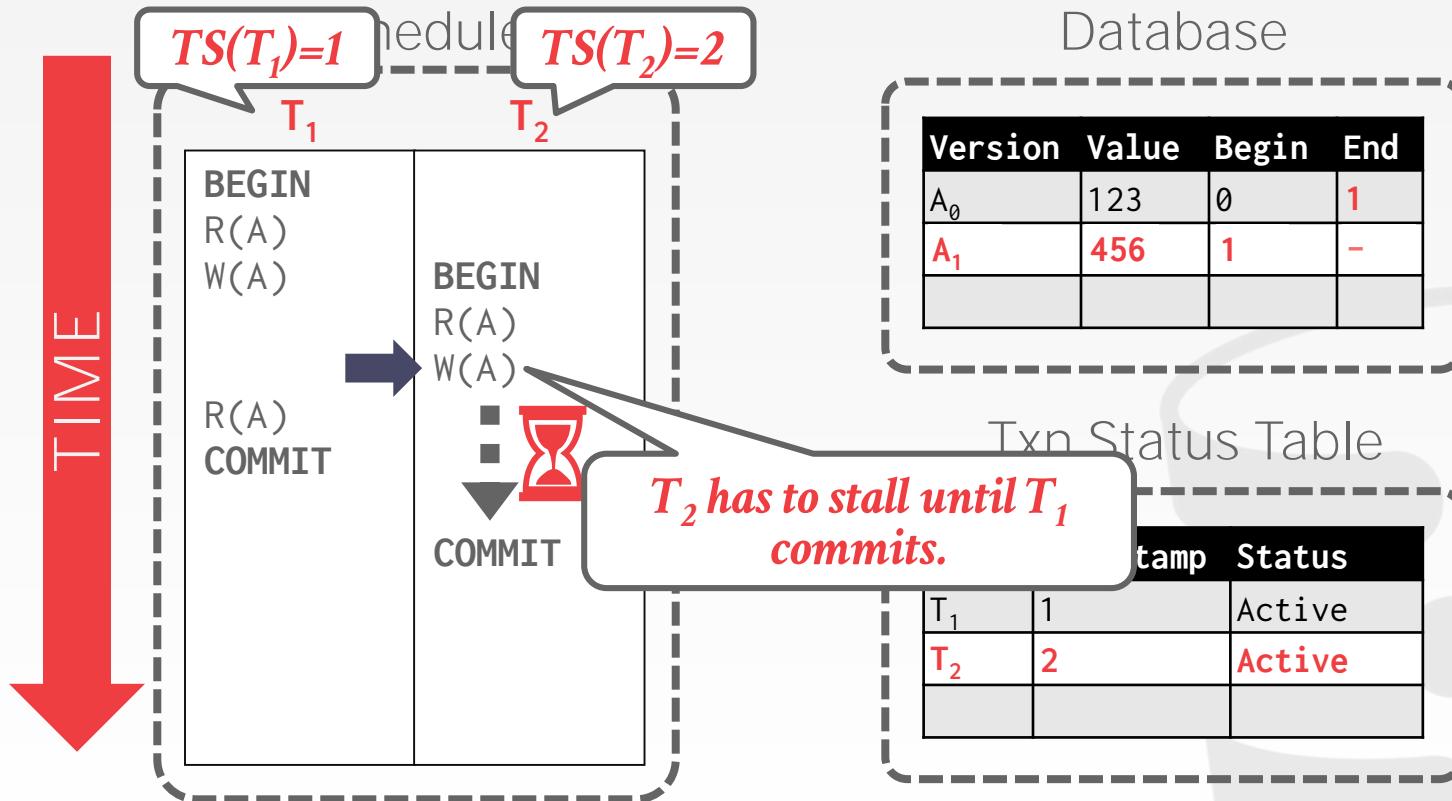
Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

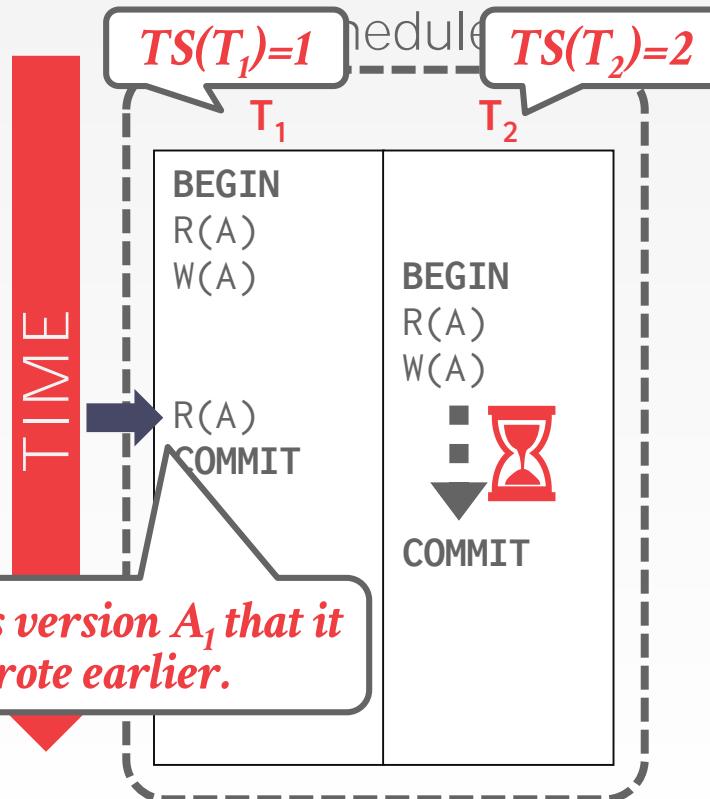
MVCC – EXAMPLE #2



MVCC – EXAMPLE #2



MVCC – EXAMPLE #2



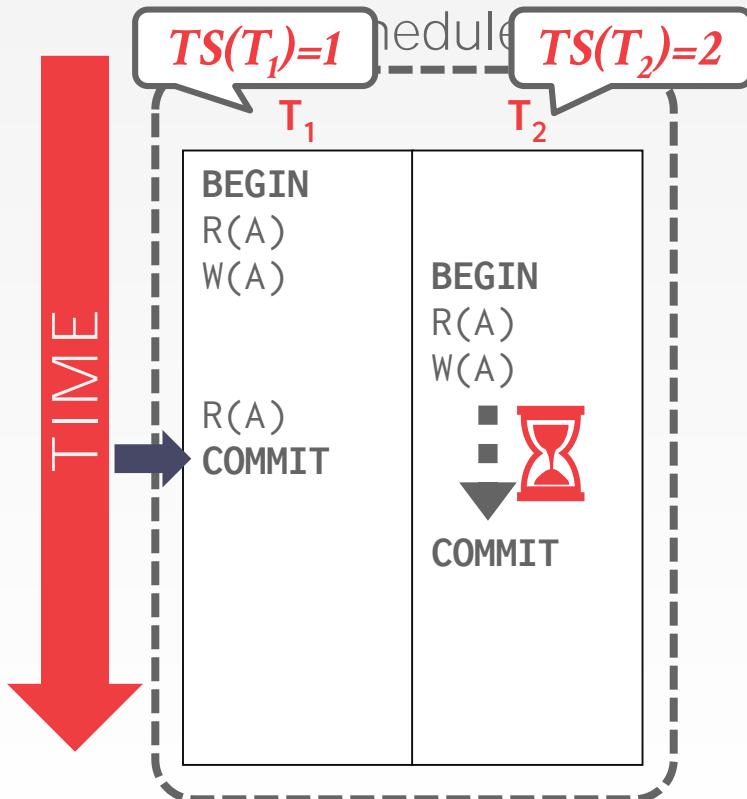
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2



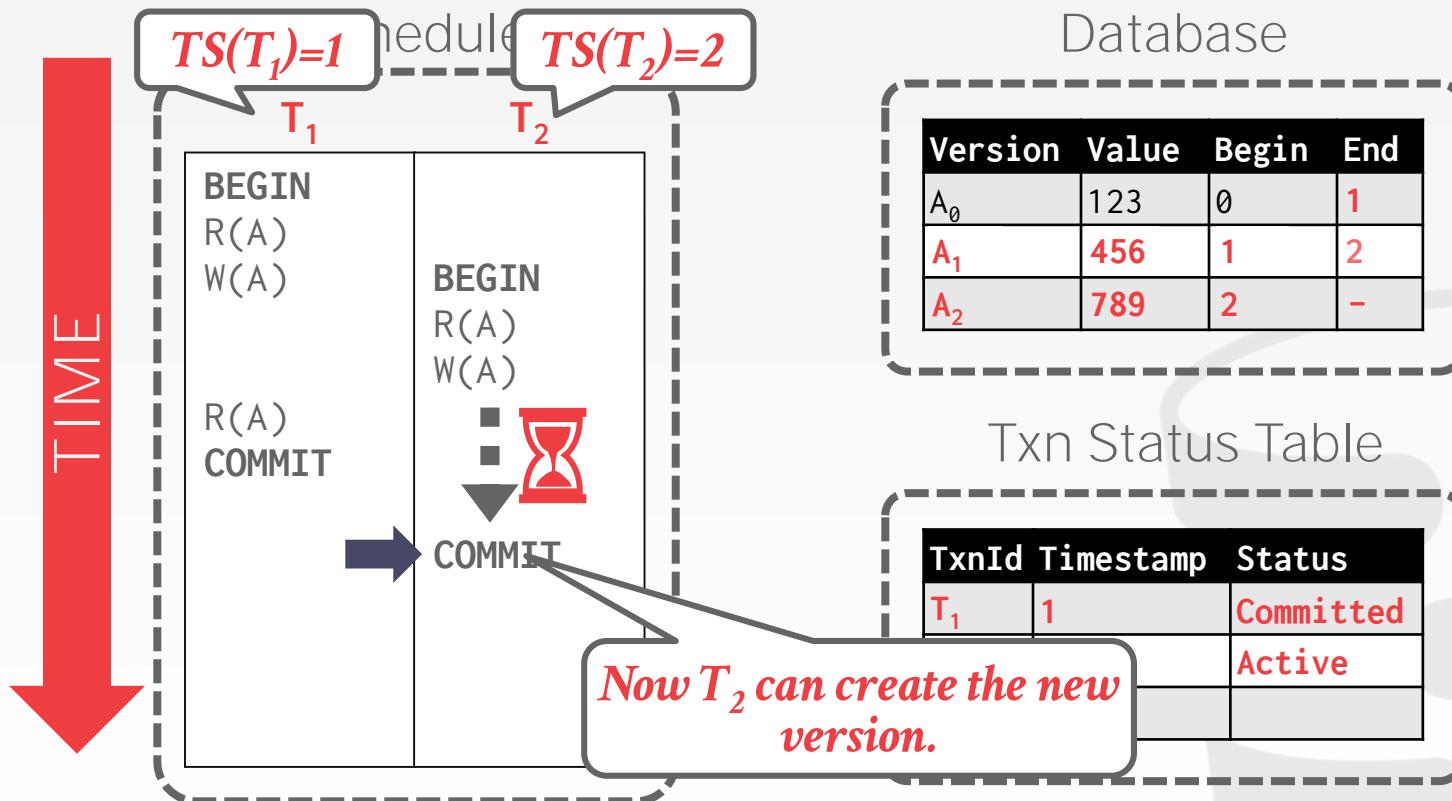
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Committed
T_2	2	Active

MVCC – EXAMPLE #2



MULTI-VERSION CONCURRENCY CONTROL

MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.



MVCC DESIGN DECISIONS

Concurrency Control Protocol

Version Storage

Garbage Collection

Index Management



CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering

- Assign txns timestamps that determine serial order.

Approach #2: Optimistic Concurrency Control

- Three-phase protocol from last class.
- Use private workspace for new versions.

Approach #3: Two-Phase Locking

- Txns acquire appropriate lock on physical version before they can read/write a logical tuple.



VERSION STORAGE

The DBMS uses the tuples' pointer field to create a **version chain** per logical tuple.

- This allows the DBMS to find the version that is visible to a particular txn at runtime.
- Indexes always point to the “head” of the chain.

Different storage schemes determine where/what to store for each version.

VERSION STORAGE

Approach #1: Append-Only Storage

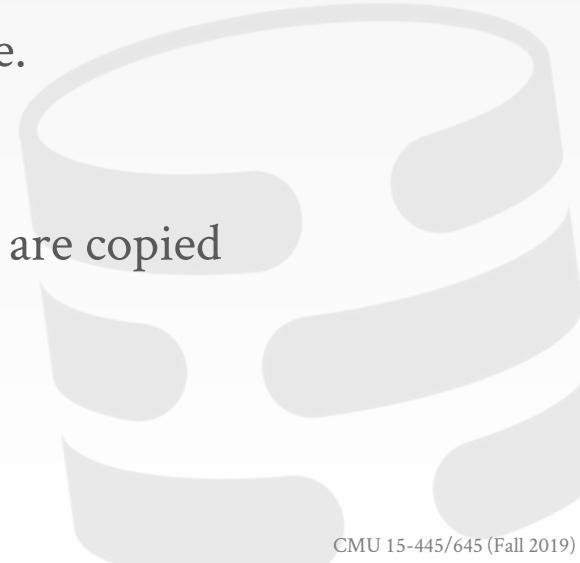
→ New versions are appended to the same table space.

Approach #2: Time-Travel Storage

→ Old versions are copied to separate table space.

Approach #3: Delta Storage

→ The original values of the modified attributes are copied into a separate delta record space.



APPEND-ONLY STORAGE

All of the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

On every update, append a new version of the tuple into an empty space in the table.

Main Table

VERSION	VALUE	POINTER
A ₀	\$111	→
A ₁	\$222	→
B ₁	\$10	→

APPEND-ONLY STORAGE

All of the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

On every update, append a new version of the tuple into an empty space in the table.

Main Table

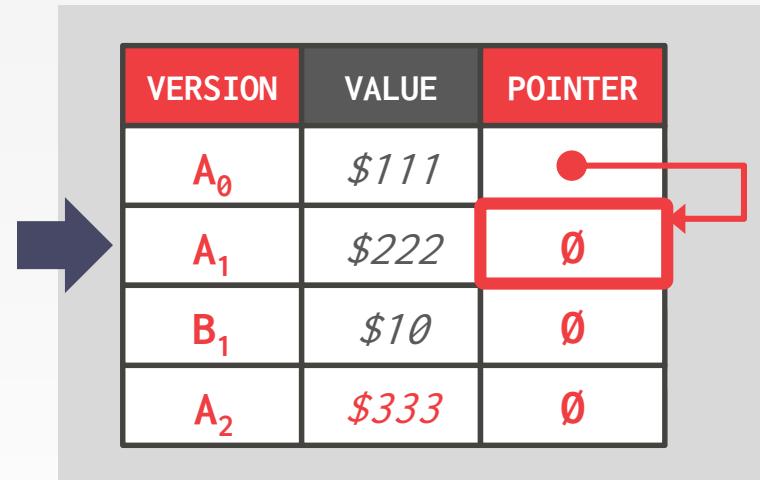
VERSION	VALUE	POINTER
A ₀	\$111	Ø
A ₁	\$222	Ø
B ₁	\$10	Ø
A ₂	\$333	Ø

APPEND-ONLY STORAGE

All of the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

On every update, append a new version of the tuple into an empty space in the table.

Main Table



VERSION	VALUE	POINTER
A ₀	\$111	•
A ₁	\$222	Ø
B ₁	\$10	Ø
A ₂	\$333	Ø

APPEND-ONLY STORAGE

All of the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

On every update, append a new version of the tuple into an empty space in the table.

Main Table

VERSION	VALUE	POINTER
A ₀	\$111	○
A ₁	\$222	○
B ₁	\$10	Ø
A ₂	\$333	Ø

VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

- Just append new version to end of the chain.
- Have to traverse chain on look-ups.

Approach #2: Newest-to-Oldest (N2O)

- Have to update index pointers for every new version.
- Don't have to traverse chain on look ups.

TIME-TRAVEL STORAGE

Main Table



VERSION	VALUE	POINTER
A ₂	\$222	
B ₁	\$10	

Time-Travel Table

VERSION	VALUE	POINTER
A ₁	\$111	

On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A ₂	\$222	
B ₁	\$10	

Time-Travel Table

VERSION	VALUE	POINTER
A ₁	\$111	
A ₂	\$222	

On every update, copy the current version to the time-travel table. Update pointers.

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A ₂	\$222	● →
B ₁	\$10	

Time-Travel Table

VERSION	VALUE	POINTER
A ₁	\$111	∅ ←
A ₂	\$222	● →

On every update, copy the current version to the time-travel table. Update pointers.

Overwrite master version in the main table.
Update pointers.

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A ₃	\$333	● →
B ₁	\$10	

Time-Travel Table

VERSION	VALUE	POINTER
A ₁	\$111	∅ ←
A ₂	\$222	● →

On every update, copy the current version to the time-travel table. Update pointers.

Overwrite master version in the main table.
Update pointers.

TIME-TRAVEL STORAGE

Main Table

VERSION	VALUE	POINTER
A ₃	\$333	
B ₁	\$10	

Time-Travel Table

VERSION	VALUE	POINTER
A ₁	\$111	
A ₂	\$222	

On every update, copy the current version to the time-travel table. Update pointers.

Overwrite master version in the main table.
Update pointers.

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A ₁	\$111	
B ₁	\$10	

Delta Storage Segment

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

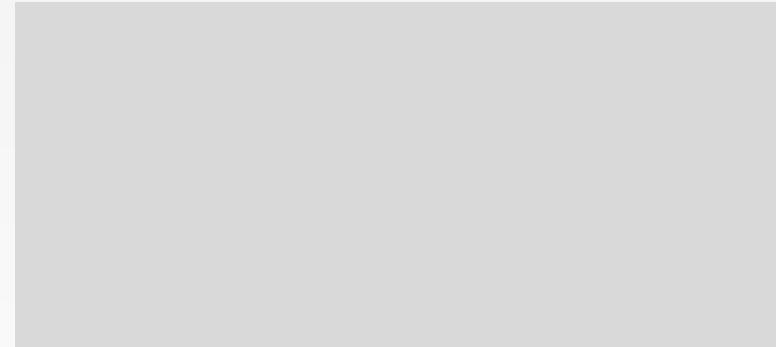
DELTA STORAGE

Main Table



VERSION	VALUE	POINTER
A ₁	\$111	
B ₁	\$10	

Delta Storage Segment



On every update, copy only the values that were modified to the delta storage and overwrite the master version.



DELTA STORAGE

Main Table



VERSION	VALUE	POINTER
A ₁	\$111	
B ₁	\$10	

Delta Storage Segment

	DELTA	POINTER
A ₁	(VALUE→\$111)	Ø

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A ₂	\$222	●
B ₁	\$10	

Delta Storage Segment

DELTA	POINTER
A ₁	(VALUE->\$111)

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A ₂	\$222	●
B ₁	\$10	

Delta Storage Segment

	DELTA	POINTER
A ₁	(VALUE->\$111)	∅
A ₂	(VALUE->\$222)	●

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table

VERSION	VALUE	POINTER
A ₃	\$333	●
B ₁	\$10	

Delta Storage Segment

	DELTA	POINTER
A ₁	(VALUE→\$111)	Ø
A ₂	(VALUE→\$222)	●

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

Txns can recreate old versions by applying the delta in reverse order.

GARBAGE COLLECTION

The DBMS needs to remove **reclaimable** physical versions from the database over time.

- No active txn in the DBMS can “see” that version (SI).
- The version was created by an aborted txn.

Two additional design decisions:

- How to look for expired versions?
- How to decide when it is safe to reclaim memory?

GARBAGE COLLECTION

Approach #1: Tuple-level

- Find old versions by examining tuples directly.
- Background Vacuuming vs. Cooperative Cleaning

Approach #2: Transaction-level

- Txns keep track of their old versions so the DBMS does not have to scan tuples to determine visibility.

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
A ₁₀₀	1	9
B ₁₀₀	1	9
B ₁₀₁	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Thread #1

$$TS(T_1)=12$$

Thread #2

$$TS(T_2)=25$$

Vacuum



VERSION	BEGIN	END
B_{101}	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Thread #1

$$TS(T_1) = 12$$

Thread #2

$$TS(T_2) = 25$$



Dirty Page BitMap

VERSION	BEGIN	END
B_{101}	10	20

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

TUPLE-LEVEL GC

Thread #1

$$TS(T_1) = 12$$

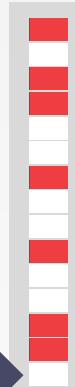
Thread #2

$$TS(T_2) = 25$$



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.



Dirty Page BitMap

VERSION	BEGIN	END
B_{101}	10	20

TUPLE-LEVEL GC

Thread #1

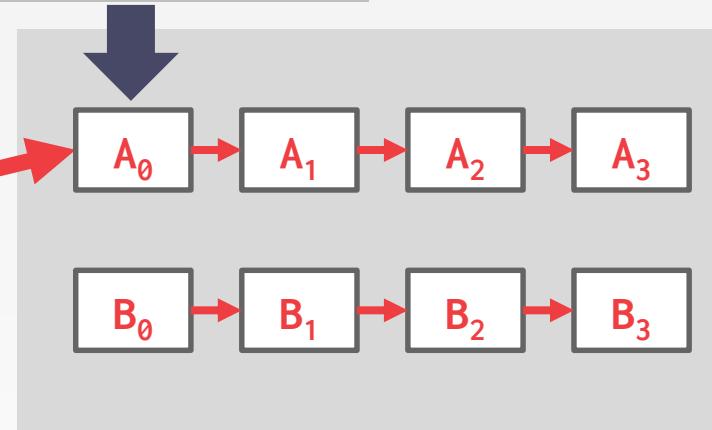
$TS(T_1)=12$



Thread #2

$TS(T_2)=25$

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.



Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC

Thread #1

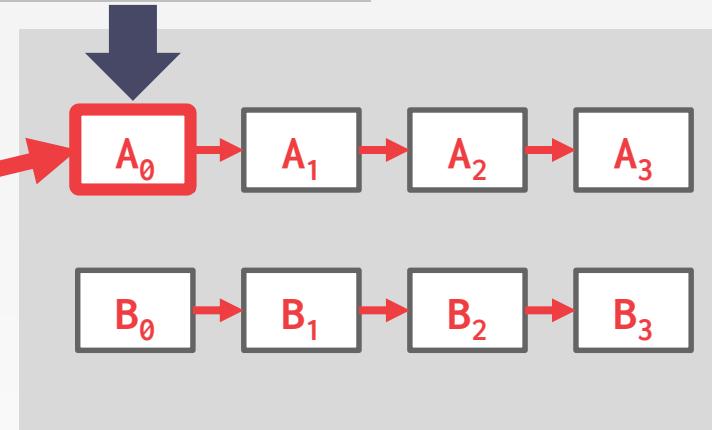
$TS(T_1)=12$



Thread #2

$TS(T_2)=25$

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.



Cooperative Cleaning:
Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

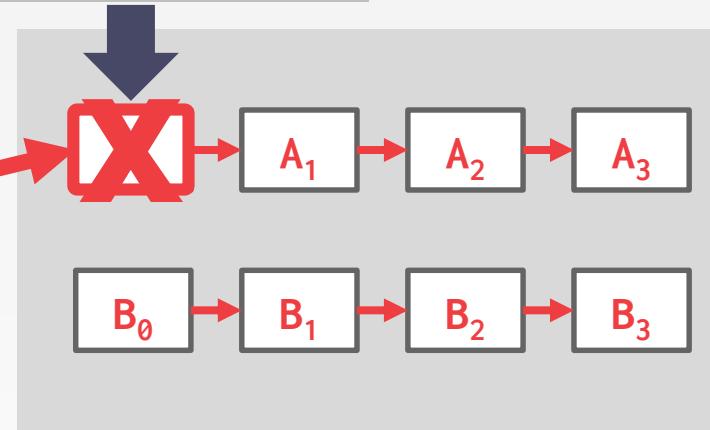
TUPLE-LEVEL GC

Thread #1

$$TS(T_1)=12$$

Thread #2

$$TS(T_2)=25$$



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:

Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

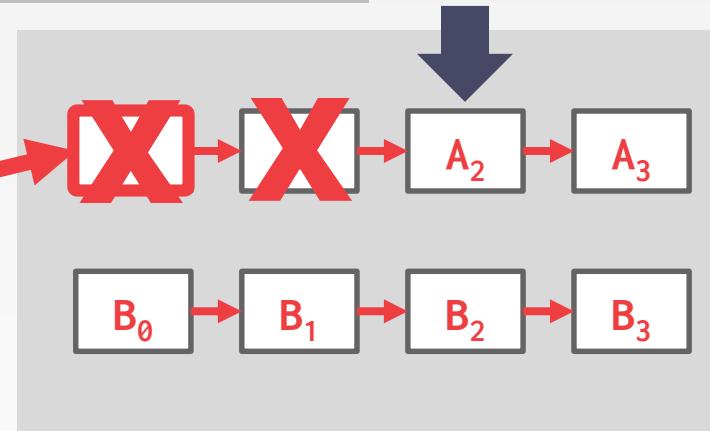
TUPLE-LEVEL GC

Thread #1

$$TS(T_1)=12$$

Thread #2

$$TS(T_2)=25$$



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:

Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

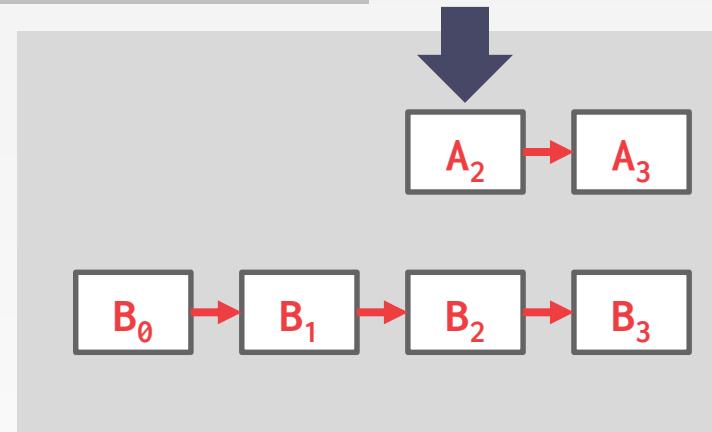


Thread #2

$TS(T_2)=25$

Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.



Cooperative Cleaning:

Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

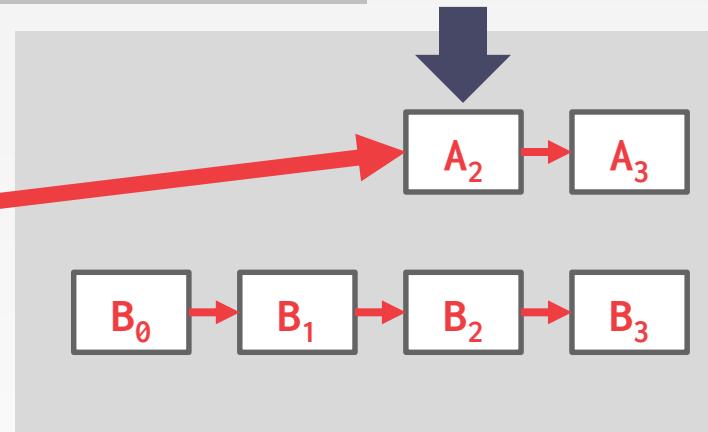
TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$



Background Vacuuming:

Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning:

Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set.

The DBMS determines when all versions created by a finished txn are no longer visible.



INDEX MANAGEMENT

Primary key indexes point to version chain head.

- How often the DBMS has to update the pkey index depends on whether the system creates new versions when a tuple is updated.
- If a txn updates a tuple's pkey attribute(s), then this is treated as an **DELETE** followed by an **INSERT**.

Secondary indexes are more complicated...

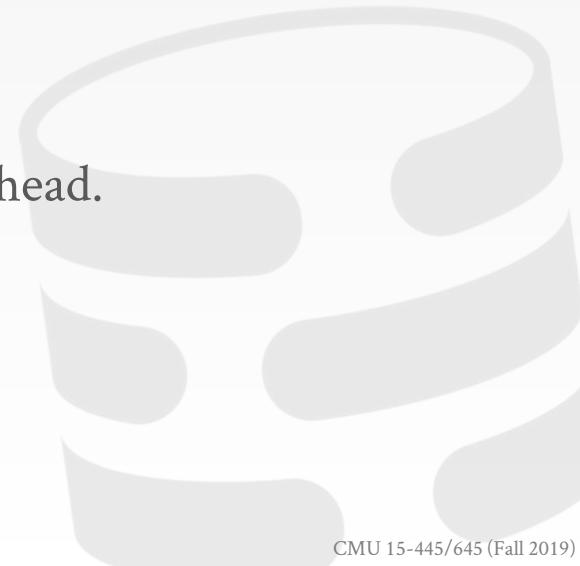
SECONDARY INDEXES

Approach #1: Logical Pointers

- Use a fixed identifier per tuple that does not change.
- Requires an extra indirection layer.
- Primary Key vs. Tuple Id

Approach #2: Physical Pointers

- Use the physical address to the version chain head.



INDEX POINTERS



PRIMARY INDEX



SECONDARY INDEX



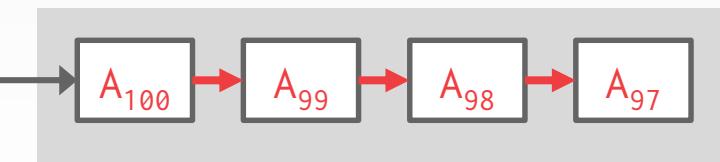
} Append-Only
Newest-to-Oldest

INDEX POINTERS

GET(A) 



Physical
Address



 } Append-Only
Newest-to-Oldest

INDEX POINTERS



PRIMARY INDEX

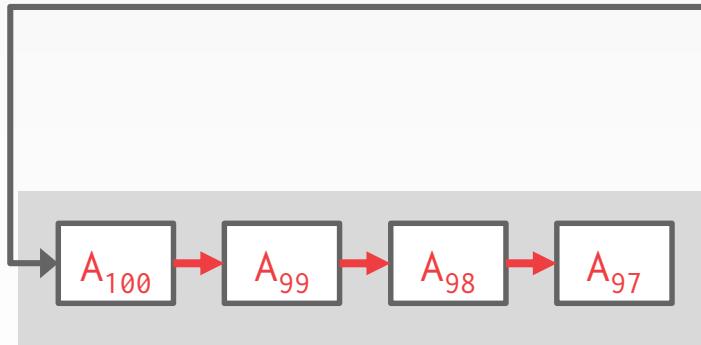


SECONDARY INDEX



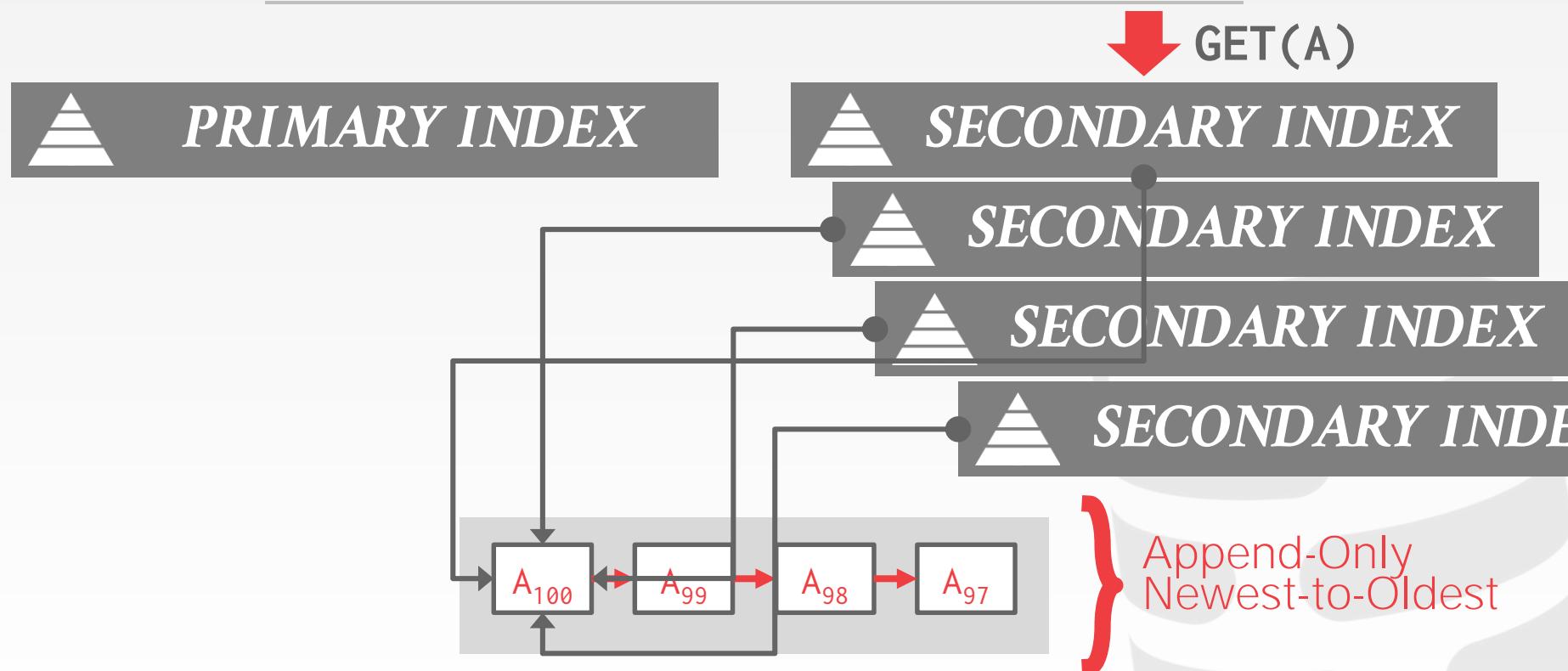
GET(A)

Physical
Address

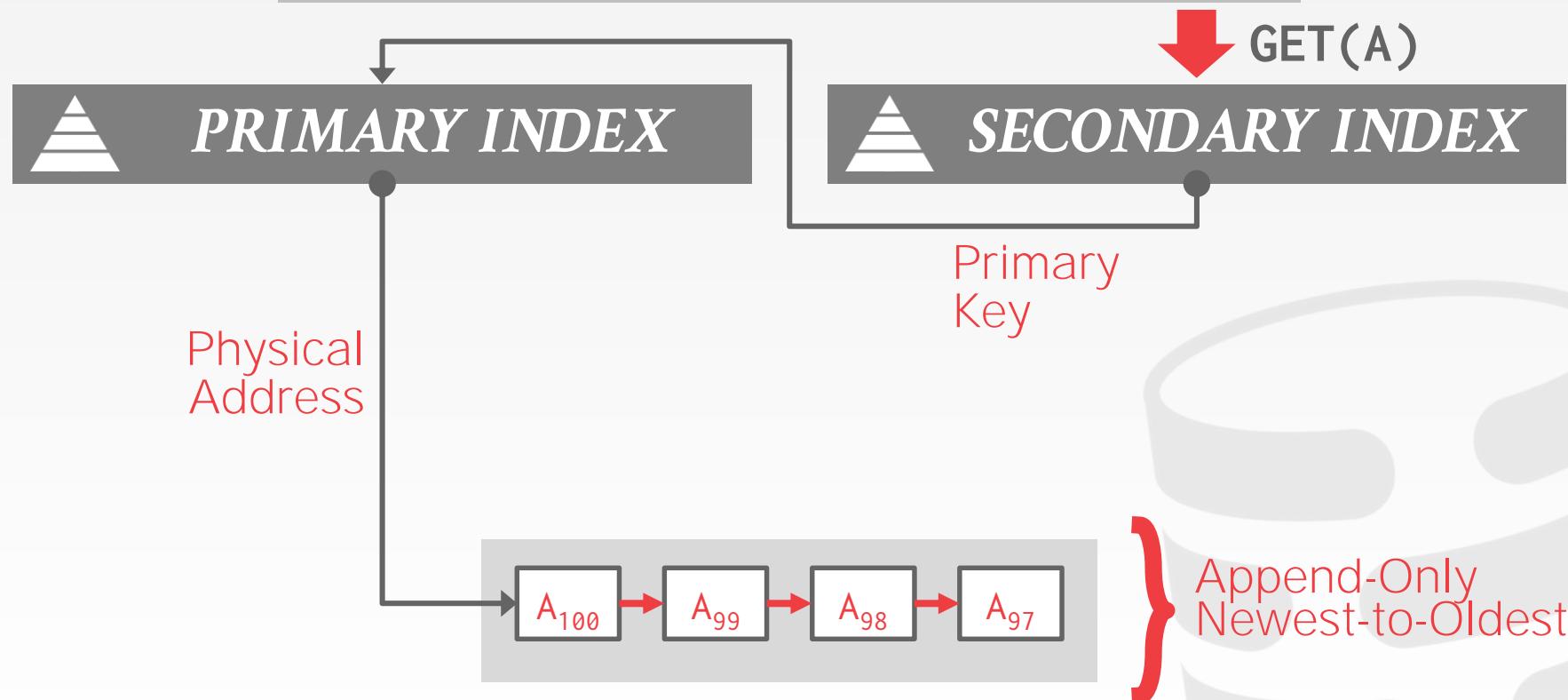


} Append-Only
Newest-to-Oldest

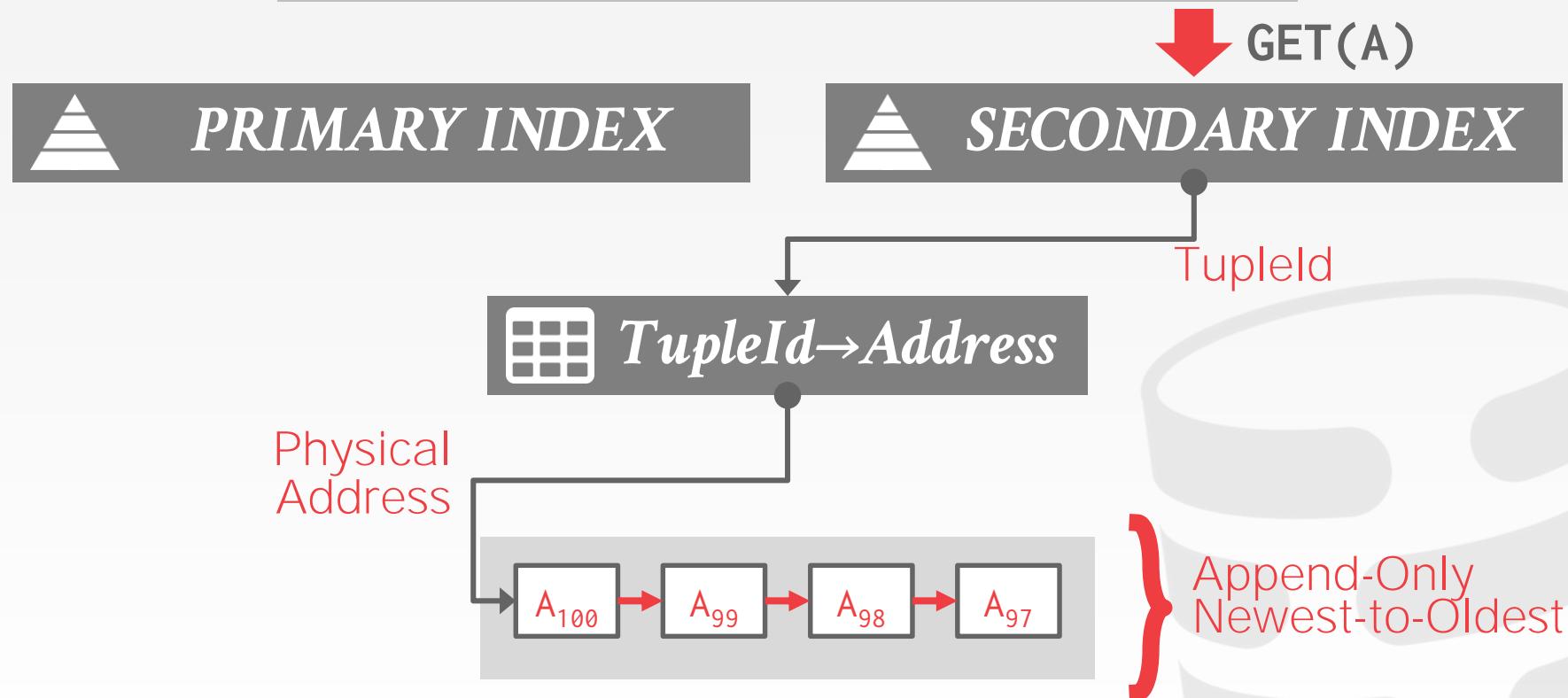
INDEX POINTERS



INDEX POINTERS



INDEX POINTERS



MVCC IMPLEMENTATIONS

	<i>Protocol</i>	<i>Version Storage</i>	<i>Garbage Collection</i>	<i>Indexes</i>
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	-	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical
<u>CMU's TBD</u>	MV-OCC	Delta	Txn-level	Logical

CONCLUSION

MVCC is the widely used scheme in DBMSs.
Even systems that do not support multi-statement
txns (e.g., NoSQL) use it.



NEXT CLASS

No class on Wed November 6th



MVCC DELETES

The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible.

- If a tuple is deleted, then there cannot be a new version of that tuple after the newest version.
- No write-write conflicts / first-writer wins

We need a way to denote that tuple has been logically delete at some point in time.



MVCC DELETES

Approach #1: Deleted Flag

- Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version.
- Can either be in tuple header or a separate column.

Approach #2: Tombstone Tuple

- Create an empty physical version to indicate that a logical tuple is deleted.
- Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce the storage overhead.

MVCC INDEXES

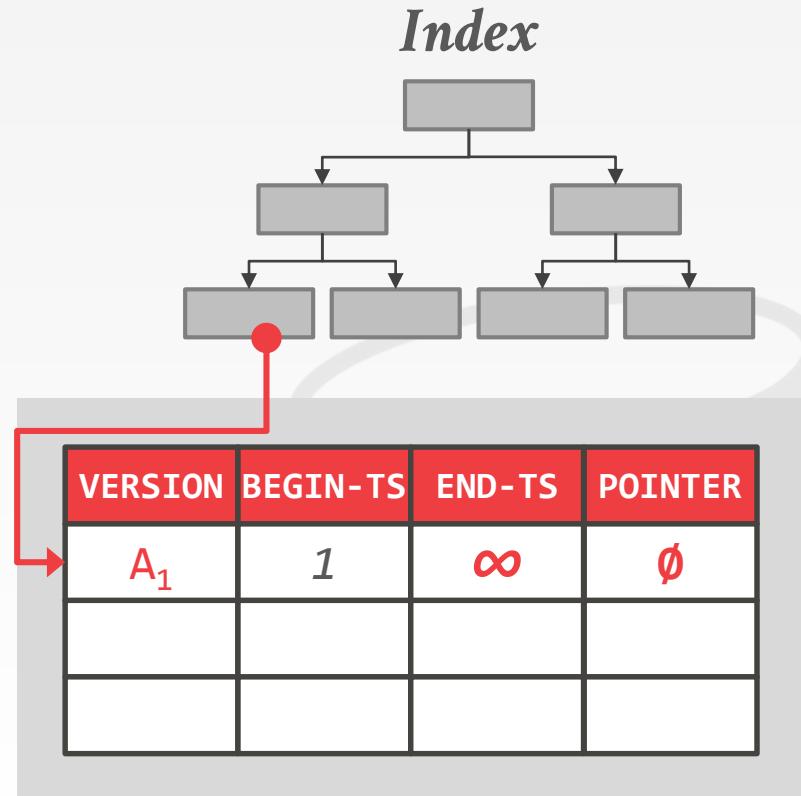
MVCC DBMS indexes (usually) do not store version information about tuples with their keys.
→ Exception: Index-organized tables (e.g., MySQL)

Every index must support duplicate keys from different snapshots:
→ The same key may point to different logical tuples in different snapshots.

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10



MVCC DUPLICATE KEY PROBLEM

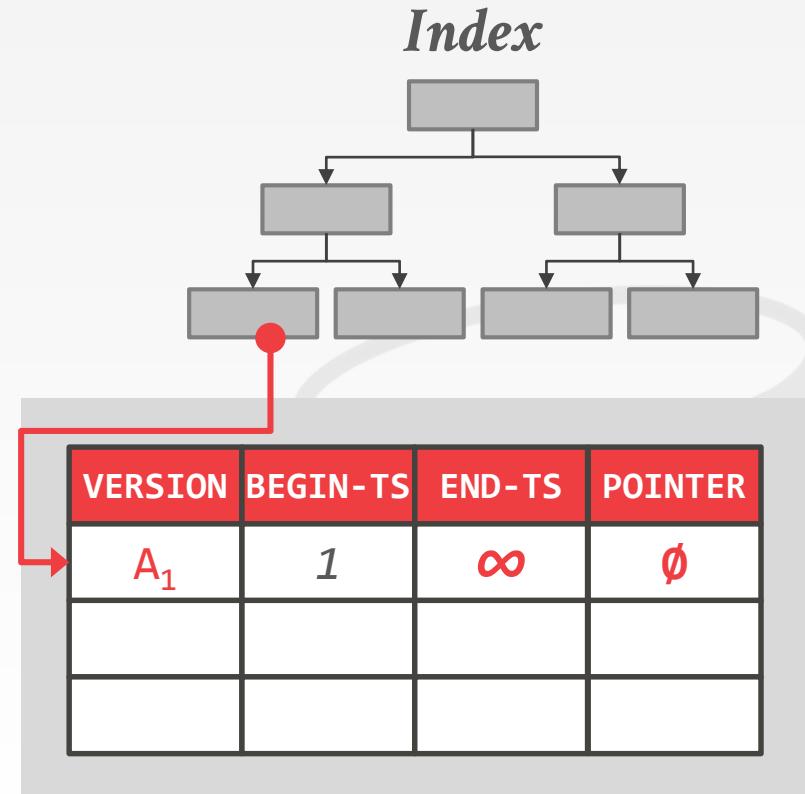
Thread #1

Begin @ 10



Thread #2

Begin @ 20



MVCC DUPLICATE KEY PROBLEM

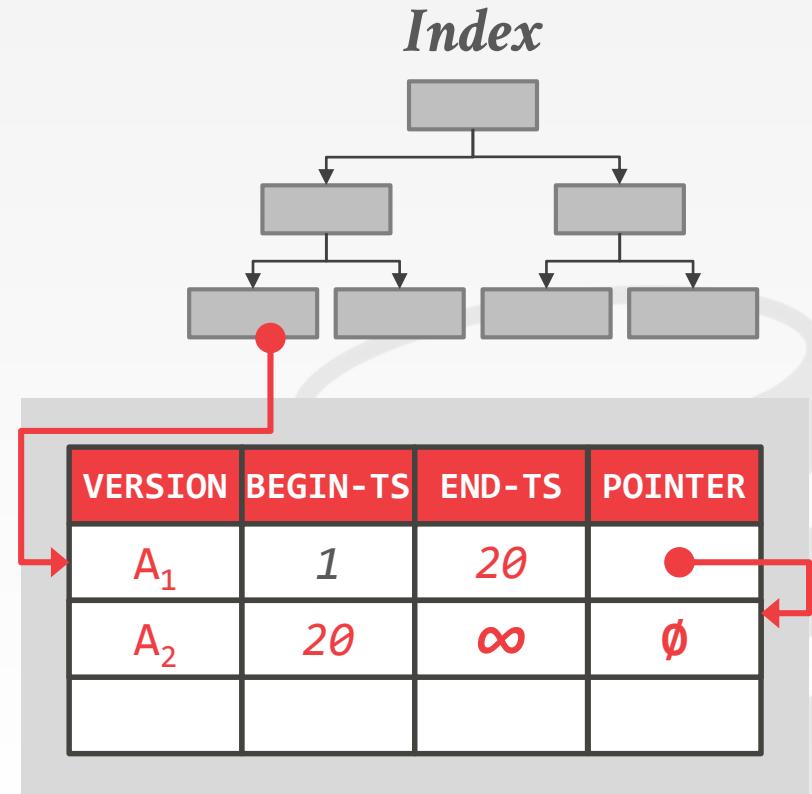
Thread #1

Begin @ 10



Thread #2

Begin @ 20



MVCC DUPLICATE KEY PROBLEM

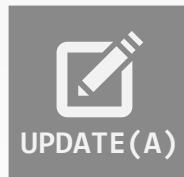
Thread #1

Begin @ 10

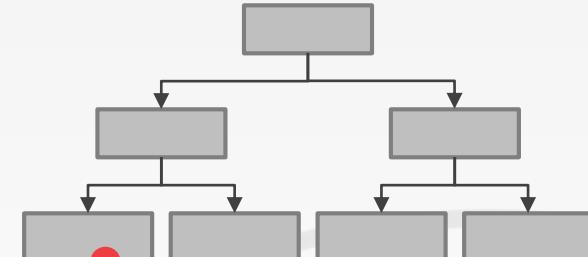


Thread #2

Begin @ 20



Index



VERSION	BEGIN-TS	END-TS	POINTER
A ₁	1	20	●
X	20	∞	∅

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10



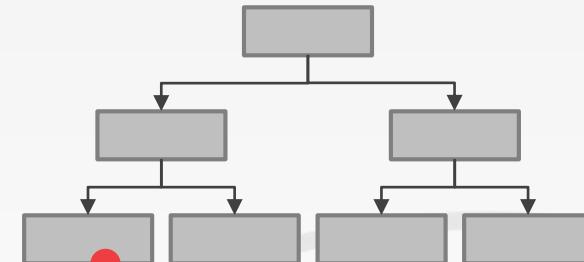
Thread #2

Begin @ 20

Commit @ 25



Index



VERSION	BEGIN-TS	END-TS	POINTER
A ₁	1	20	→
X	20	∞	∅

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10



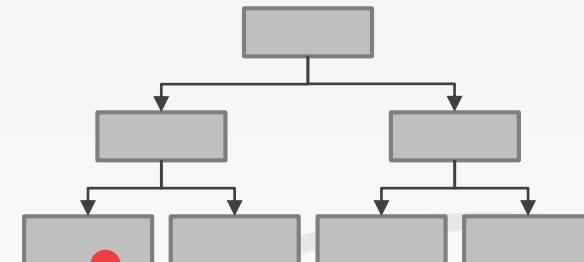
Thread #2

Begin @ 20

Commit @ 25



Index



VERSION	BEGIN-TS	END-TS	POINTER
A ₁	1	25	→
X	25	25	∅

MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10



Thread #2

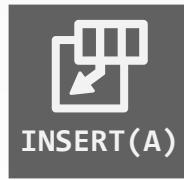
Begin @ 20

Commit @ 25

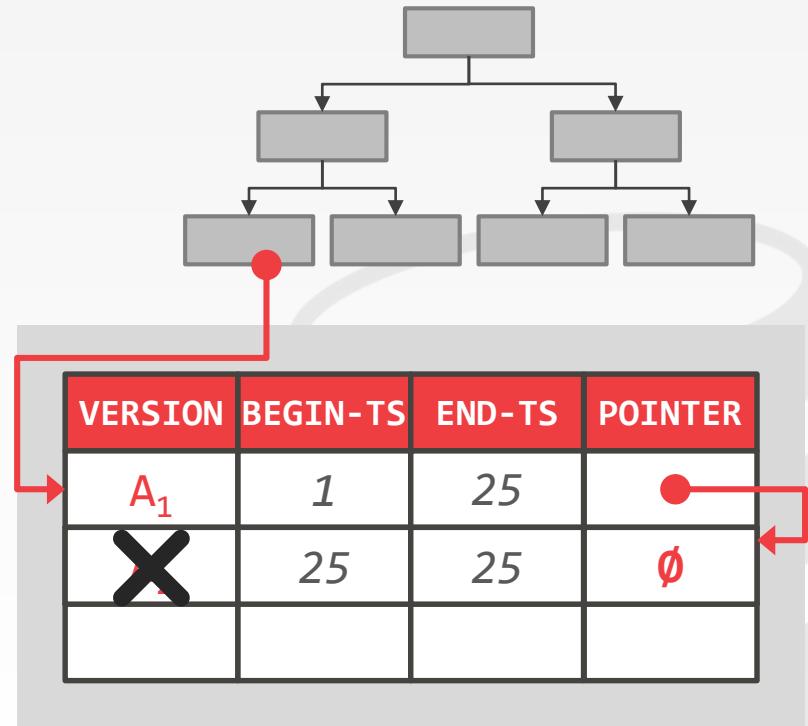


Thread #3

Begin @ 30



Index



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10



Thread #2

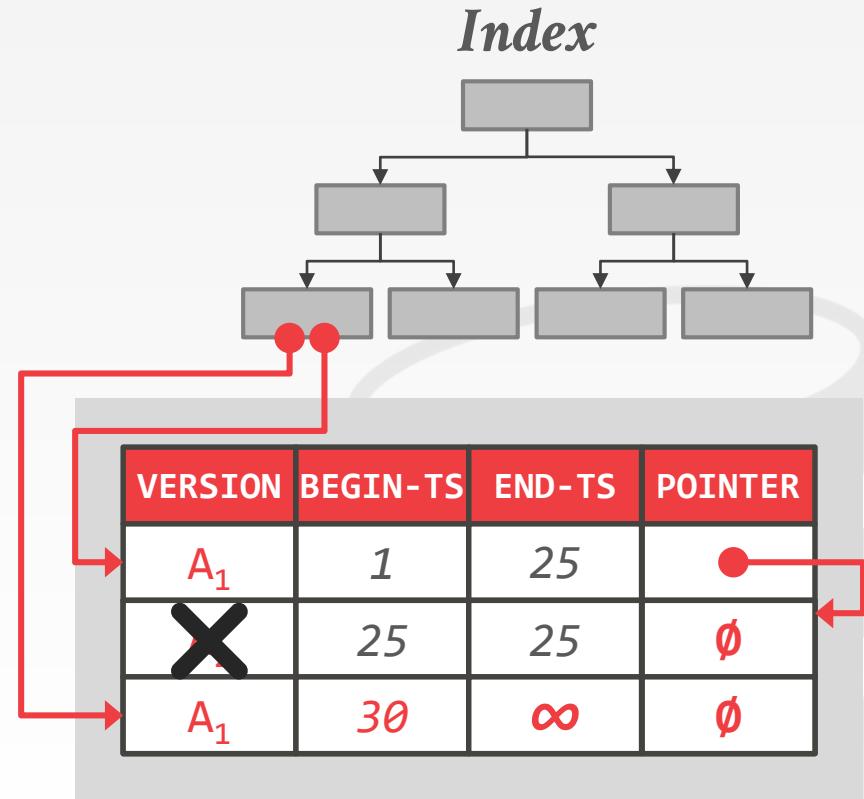
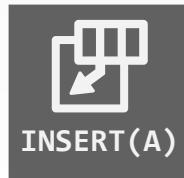
Begin @ 20

Commit @ 25



Thread #3

Begin @ 30



MVCC DUPLICATE KEY PROBLEM

Thread #1

Begin @ 10



Thread #2

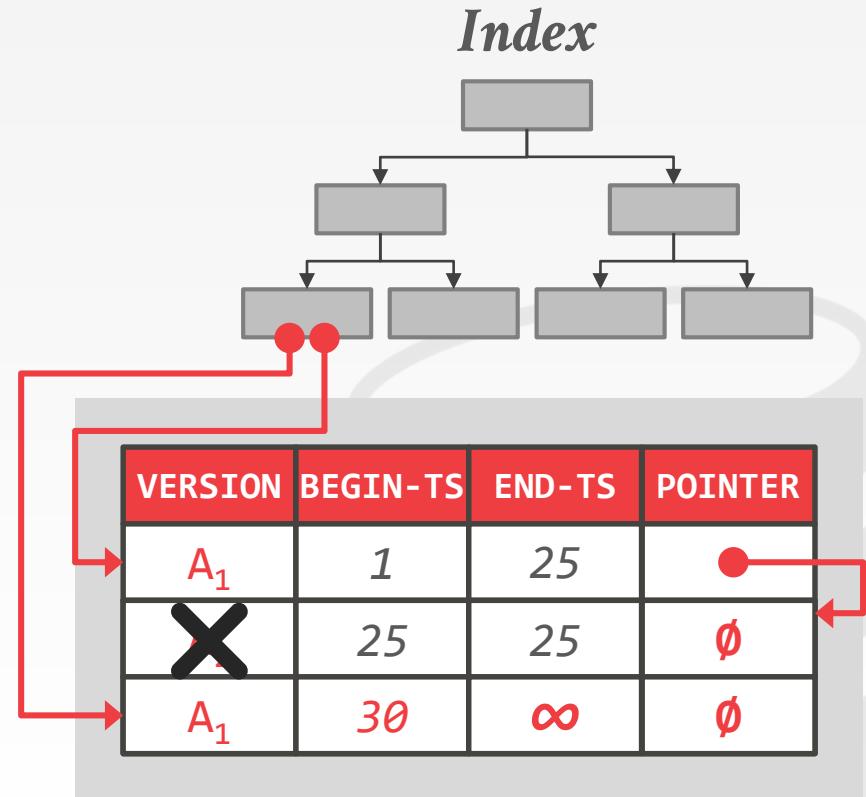
Begin @ 20

Commit @ 25



Thread #3

Begin @ 30



MVCC INDEXES

Each index's underlying data structure has to support the storage of non-unique keys.

Use additional execution logic to perform conditional inserts for pkey / unique indexes.
→ Atomically check whether the key exists and then insert.

Workers may get back multiple entries for a single fetch. They then have to follow the pointers to find the proper physical version.

20|

Logging Schemes



Intro to Database Systems
15-445/15-645
Fall 2019

AP

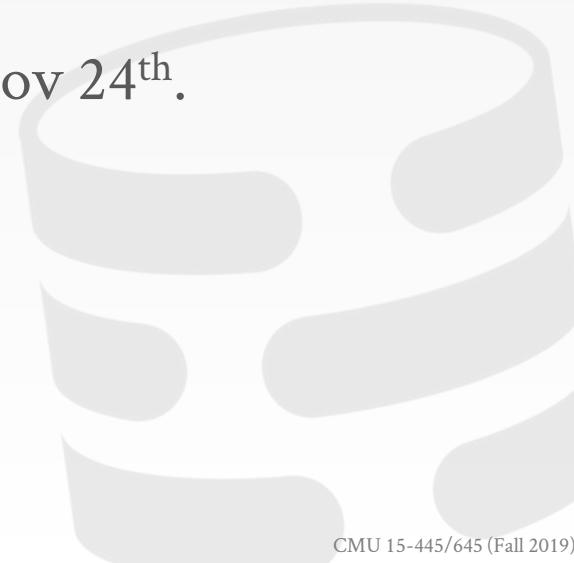
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

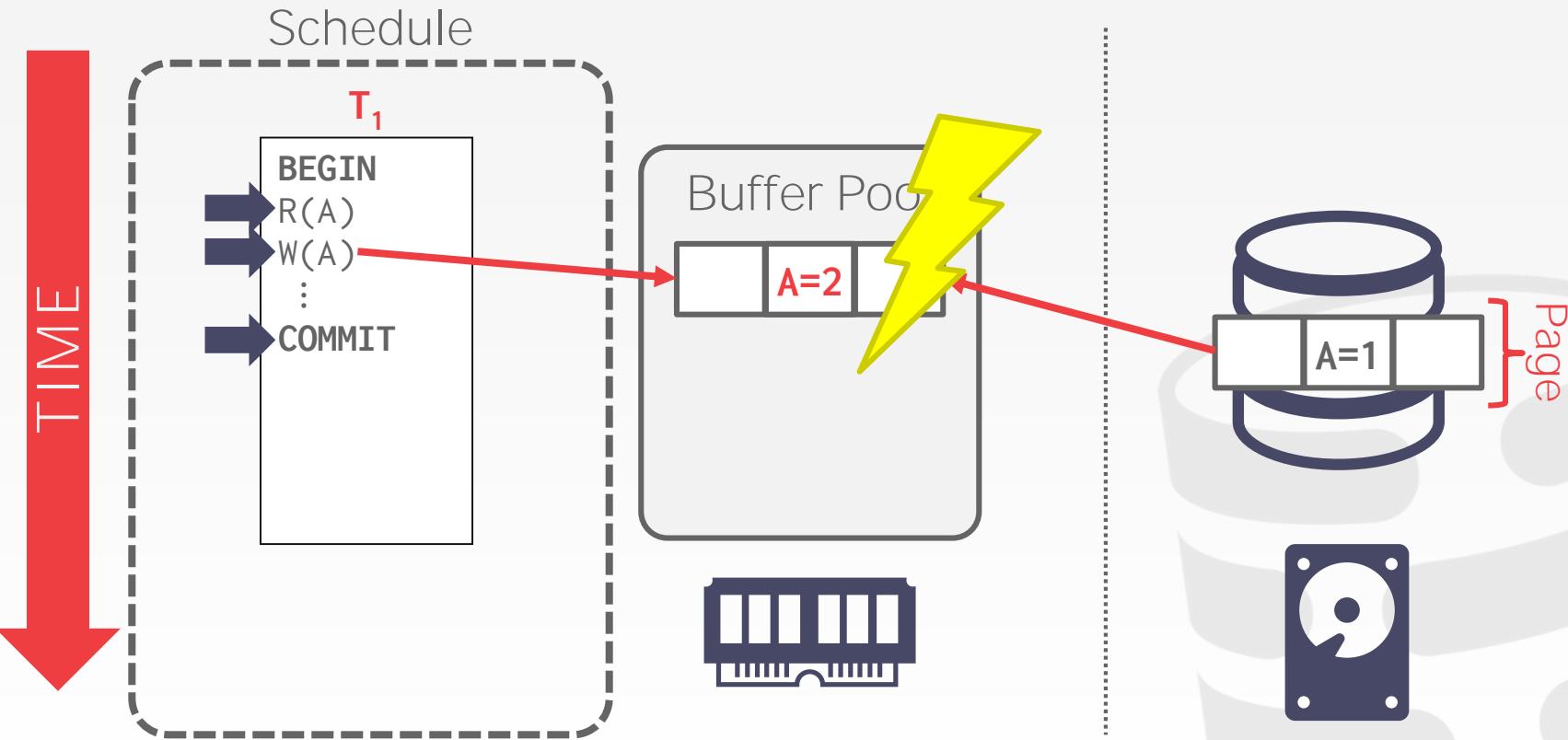
Homework #4 is due Wed Nov 13th @ 11:59pm.

Project #3 is due Sun Nov 17th @ 11:59pm.

Extra Credit Checkpoint is due Sun Nov 24th.



MOTIVATION



CRASH RECOVERY

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures.

Recovery algorithms have two parts:

- Actions during normal txn processing to ensure that the DBMS can recover from a failure.
- Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

Today

TODAY'S AGENDA

Failure Classification

Buffer Pool Policies

Shadow Paging

Write-Ahead Log

Logging Schemes

Checkpoints



CRASH RECOVERY

DBMS is divided into different components based on the underlying storage device.

We must also classify the different types of failures that the DBMS needs to handle.



FAILURE CLASSIFICATION

Type #1 – Transaction Failures

Type #2 – System Failures

Type #3 – Storage Media Failures



TRANSACTION FAILURES

Logical Errors:

- Transaction cannot complete due to some internal error condition (e.g., integrity constraint violation).

Internal State Errors:

- DBMS must terminate an active transaction due to an error condition (e.g., deadlock).

SYSTEM FAILURES

Software Failure:

- Problem with the DBMS implementation (e.g., uncaught divide-by-zero exception).

Hardware Failure:

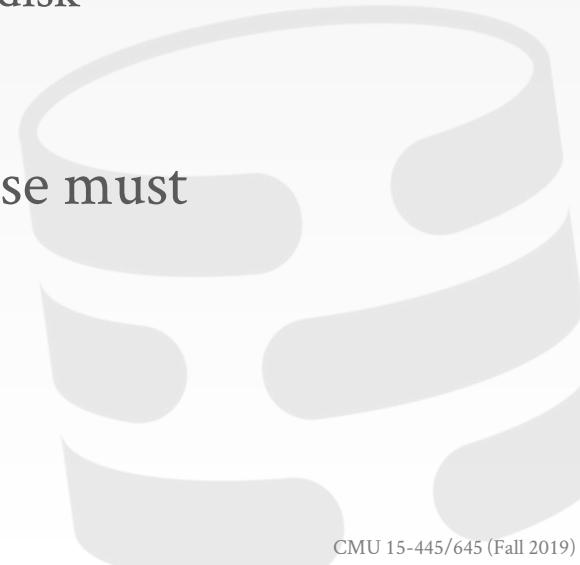
- The computer hosting the DBMS crashes (e.g., power plug gets pulled).
- Fail-stop Assumption: Non-volatile storage contents are assumed to not be corrupted by system crash.

STORAGE MEDIA FAILURE

Non-Repairable Hardware Failure:

- A head crash or similar disk failure destroys all or part of non-volatile storage.
- Destruction is assumed to be detectable (e.g., disk controller use checksums to detect failures).

No DBMS can recover from this! Database must be restored from archived version.



OBSERVATION

The primary storage location of the database is on non-volatile storage, but this is much slower than volatile storage.

Use volatile memory for faster access:

- First copy target record into memory.
- Perform the writes in memory.
- Write dirty records back to disk.



OBSERVATION

The DBMS needs to ensure the following guarantees:

- The changes for any txn are durable once the DBMS has told somebody that it committed.
- No partial changes are durable if the txn aborted.



UNDO VS. REDO

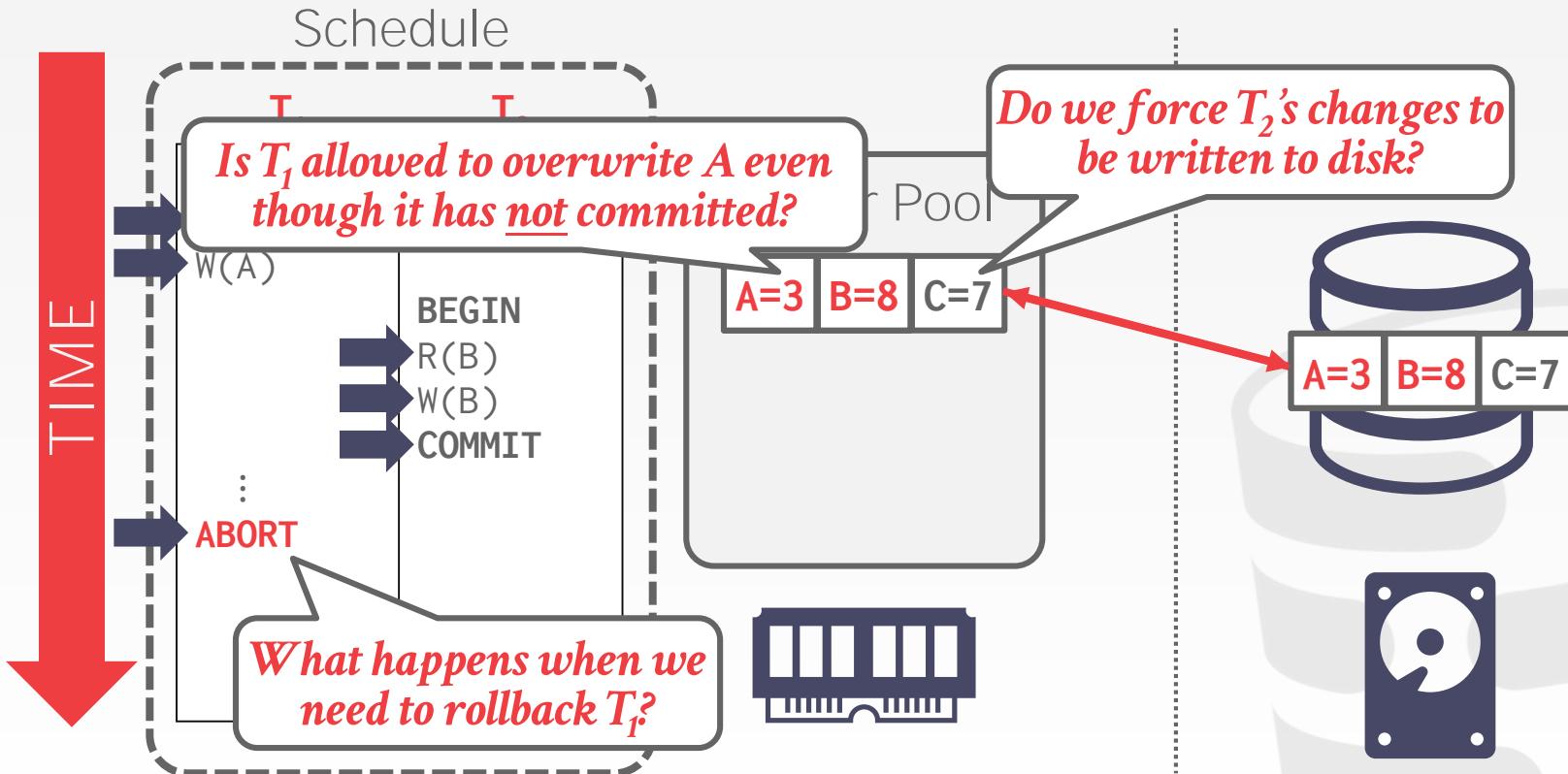
Undo: The process of removing the effects of an incomplete or aborted txn.

Redo: The process of re-instantating the effects of a committed txn for durability.

How the DBMS supports this functionality depends on how it manages the buffer pool...



BUFFER POOL



STEAL POLICY

Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage.

STEAL: Is allowed.

NO-STEAL: Is not allowed.



FORCE POLICY

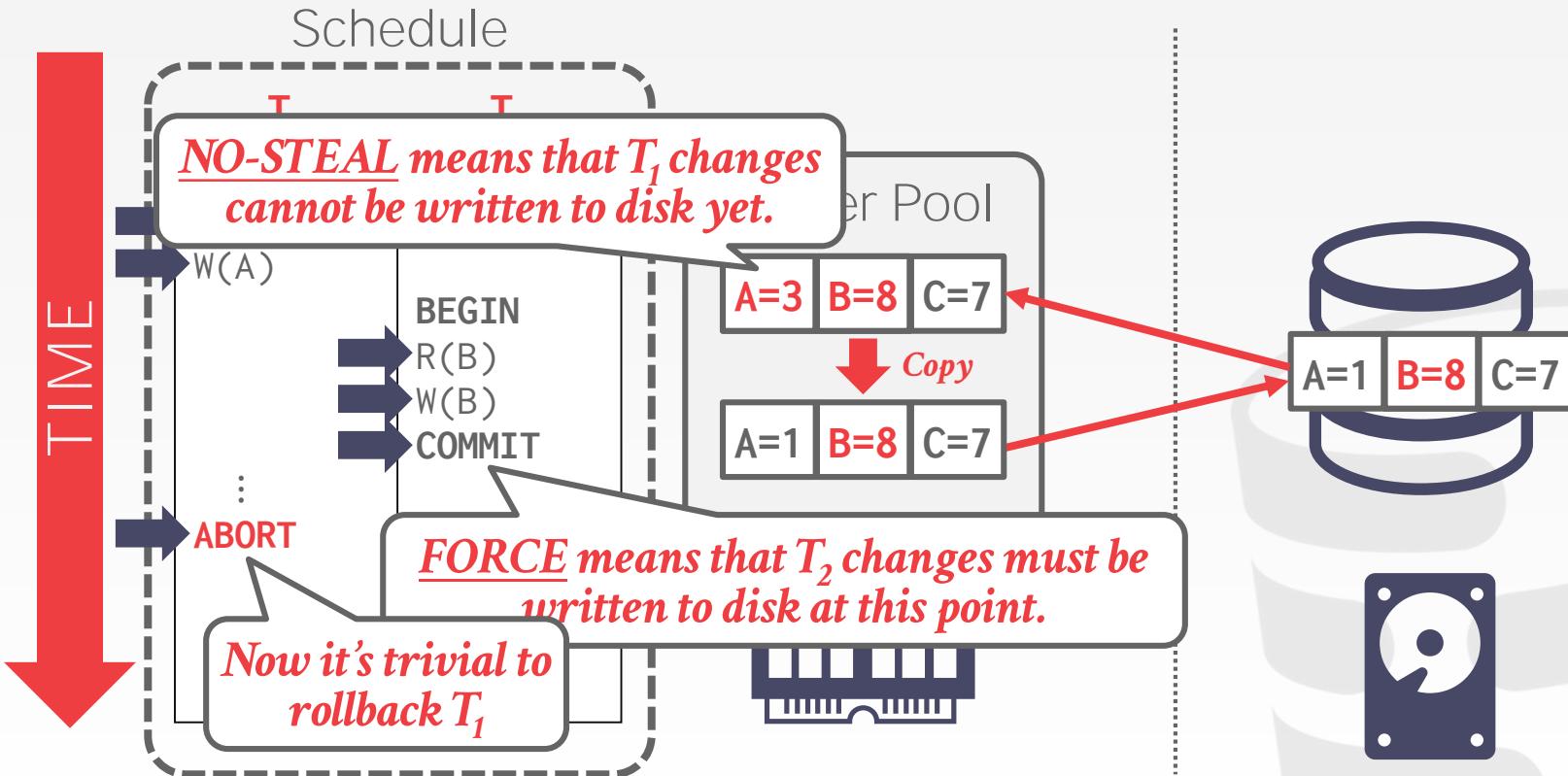
Whether the DBMS requires that all updates made by a txn are reflected on non-volatile storage before the txn is allowed to commit.

FORCE: Is required.

NO-FORCE: Is not required.



NO-STEAL + FORCE



NO-STEAL + FORCE

This approach is the easiest to implement:

- Never have to undo changes of an aborted txn because the changes were not written to disk.
- Never have to redo changes of a committed txn because all the changes are guaranteed to be written to disk at commit time (assuming atomic hardware writes).

Previous example cannot support write sets that exceed the amount of physical memory available.

SHADOW PAGING

Maintain two separate copies of the database:

- **Master**: Contains only changes from committed txns.
- **Shadow**: Temporary database with changes made from uncommitted txns.

Txns only make updates in the shadow copy.

When a txn commits, atomically switch the shadow to become the new master.

Buffer Pool Policy: **NO-STEAL + FORCE**

SHADOW PAGING

Instead of copying the entire database, the DBMS copies pages on write.

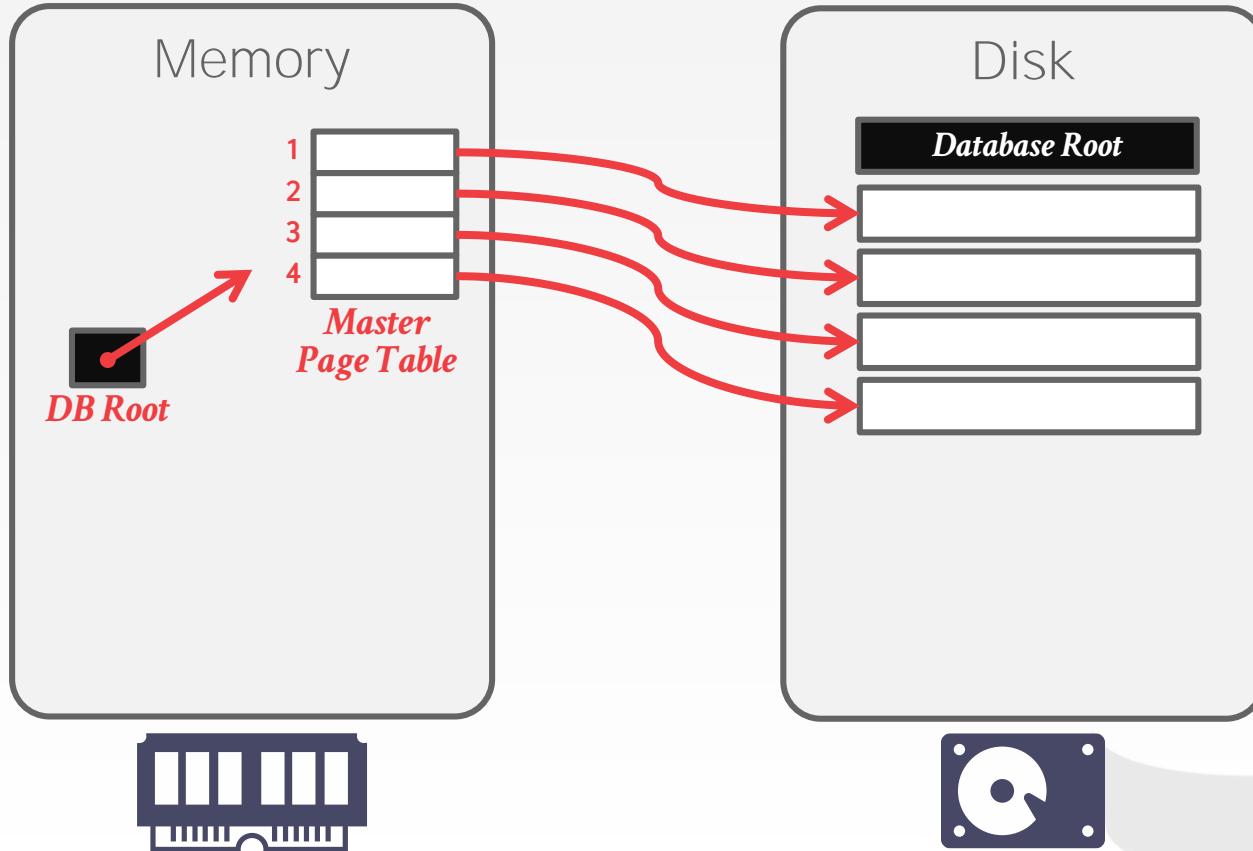
Organize the database pages in a tree structure where the root is a single disk page.

There are two copies of the tree, the master and shadow

- The root points to the master copy.
- Updates are applied to the shadow copy.



SHADOW PAGING – EXAMPLE



SHADOW PAGING

To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:

- Before overwriting the root, none of the txn's updates are part of the disk-resident database
- After overwriting the root, all the txn's updates are part of the disk-resident database.

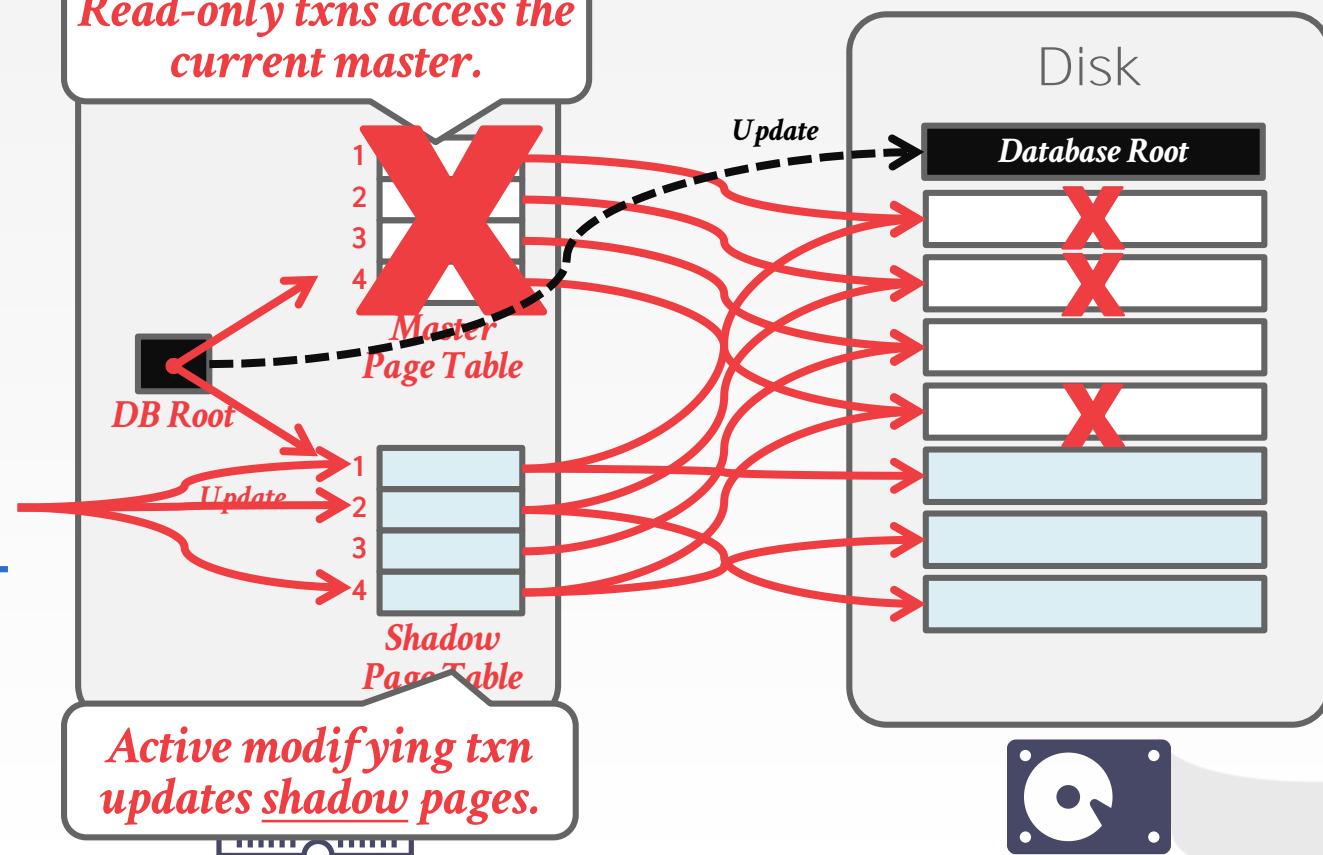


Source: [The Great Phil Bernstein](#)

SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

COMMIT



SHADOW PAGING – UNDO/REDO

Supporting rollbacks and recovery is easy.

Undo: Remove the shadow pages. Leave the master and the DB root pointer alone.

Redo: Not needed at all.



SHADOW PAGING – DISADVANTAGES

Copying the entire page table is expensive:

- Use a page table structured like a B+tree.
- No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

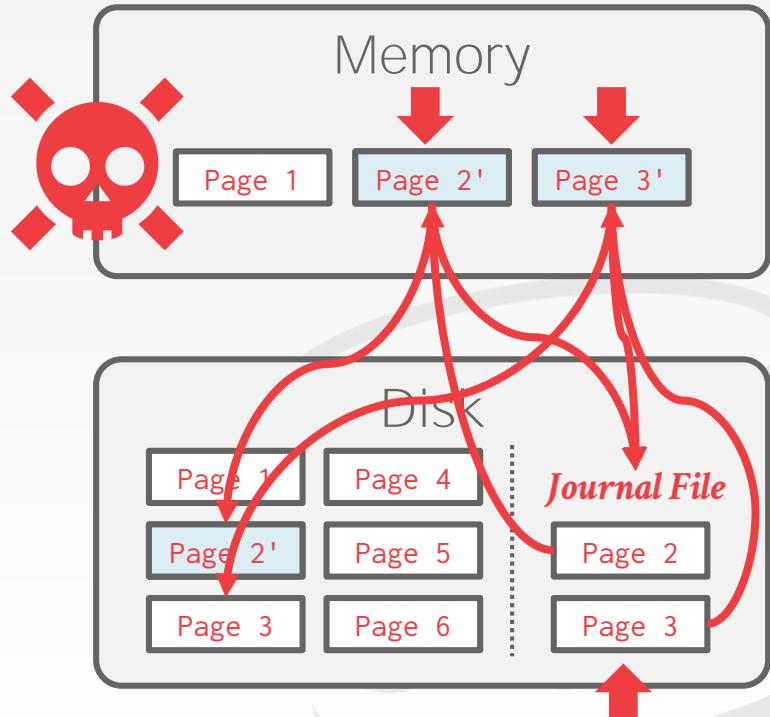
Commit overhead is high:

- Flush every updated page, page table, and root.
- Data gets fragmented.
- Need garbage collection.
- Only supports one writer txn at a time or txns in a batch.

SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



OBSERVATION

Shadowing page requires the DBMS to perform writes to random non-contiguous pages on disk.

We need a way for the DBMS convert random writes into sequential writes.



WRITE-AHEAD LOG

Maintain a log file separate from data files that contains the changes that txns make to database.

- Assume that the log is on stable storage.
- Log contains enough information to perform the necessary undo and redo actions to restore the database.

DBMS must write to disk the log file records that correspond to changes made to a database object **before** it can flush that object to disk.

Buffer Pool Policy: **STEAL + NO-FORCE**

WAL PROTOCOL

The DBMS stages all a txn's log records in volatile storage (usually backed by buffer pool).

All log records pertaining to an updated page are written to non-volatile storage before the page itself is over-written in non-volatile storage.

A txn is not considered committed until all its log records have been written to stable storage.

WAL PROTOCOL

Write a <**BEGIN**> record to the log for each txn to mark its starting point.

When a txn finishes, the DBMS will:

- Write a <**COMMIT**> record on the log
- Make sure that all log records are flushed before it returns an acknowledgement to application.



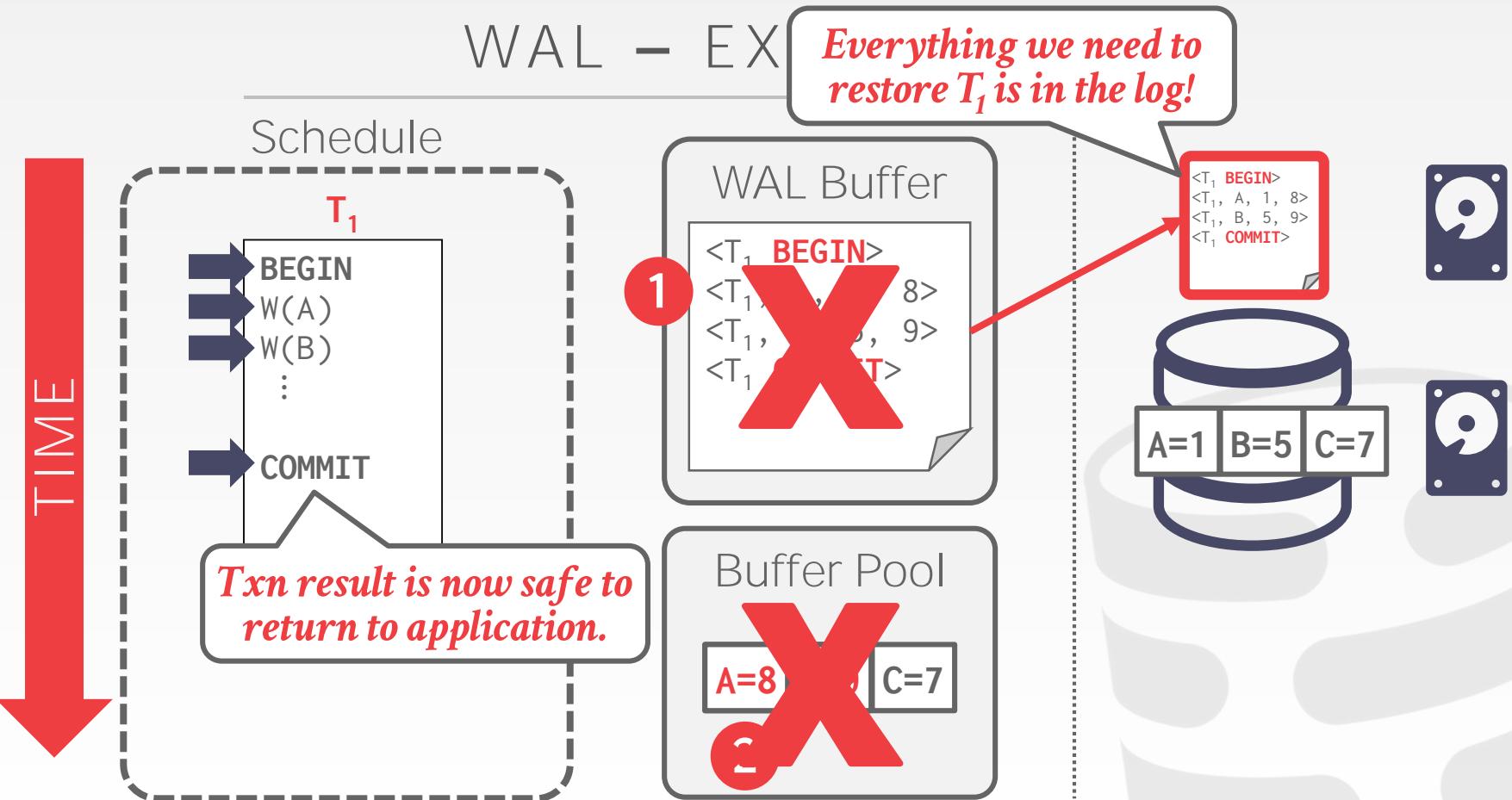
WAL PROTOCOL

Each log entry contains information about the change to a single object:

- Transaction Id
- Object Id
- Before Value (UNDO)
- After Value (REDO)



Everything we need to restore T_1 is in the log!



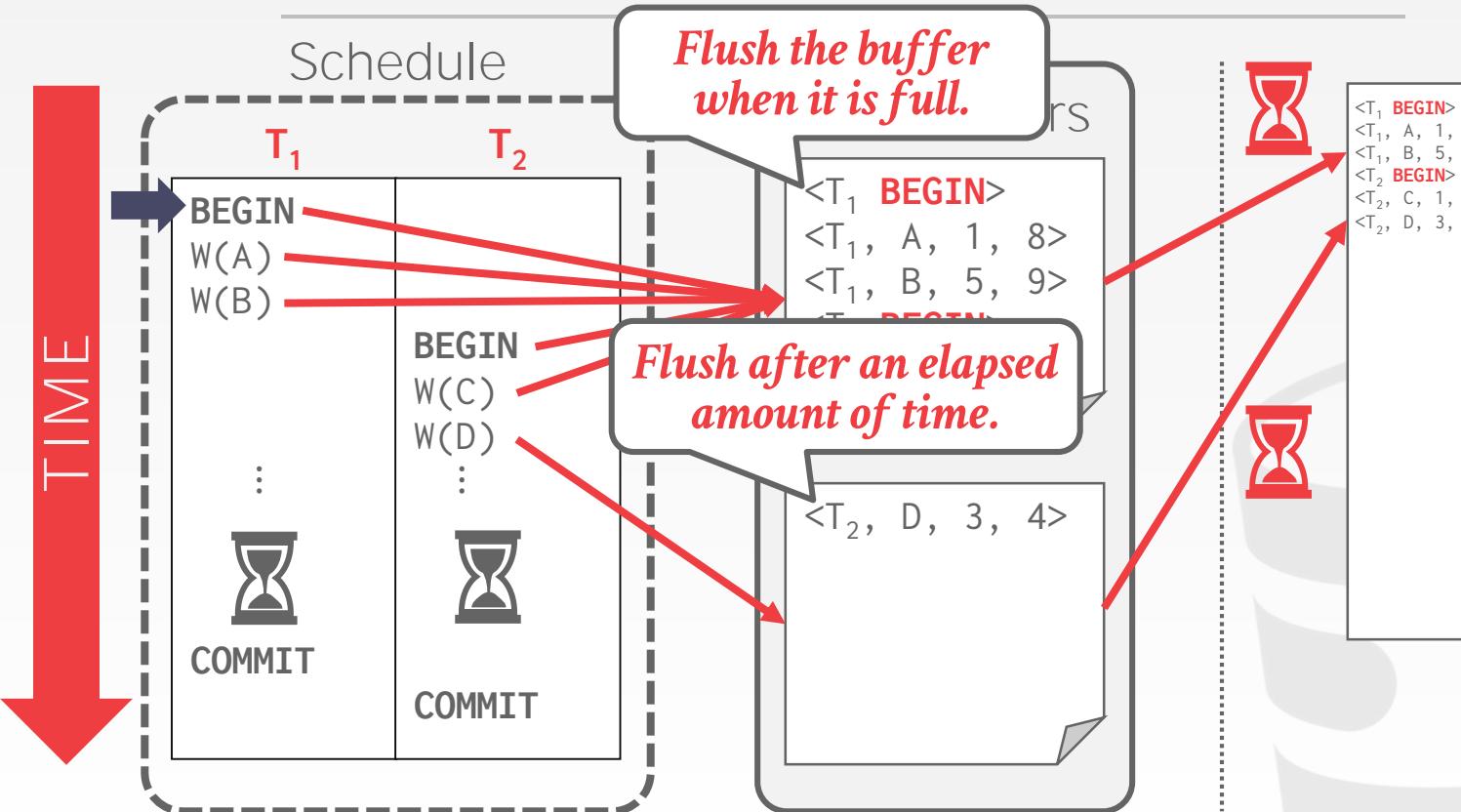
WAL – IMPLEMENTATION

When should the DBMS write log entries to disk?

- When the transaction commits.
- Can use group commit to batch multiple log flushes together to amortize overhead.



WAL - GROUP COMMIT



WAL – IMPLEMENTATION

When should the DBMS write log entries to disk?

- When the transaction commits.
- Can use group commit to batch multiple log flushes together to amortize overhead.

When should the DBMS write dirty records to disk?

- Every time the txn executes an update?
- Once when the txn commits?



BUFFER POOL POLICIES

Runtime Performance

	NO-STEAL	STEAL
NO-FORCE	-	Fastest
FORCE	Slowest	-

Recovery Performance

	NO-STEAL	STEAL
NO-FORCE	-	Slowest
FORCE	Fastest	-

Undo + Redo

No Undo + No Redo

Almost every DBMS uses **NO-FORCE + STEAL**

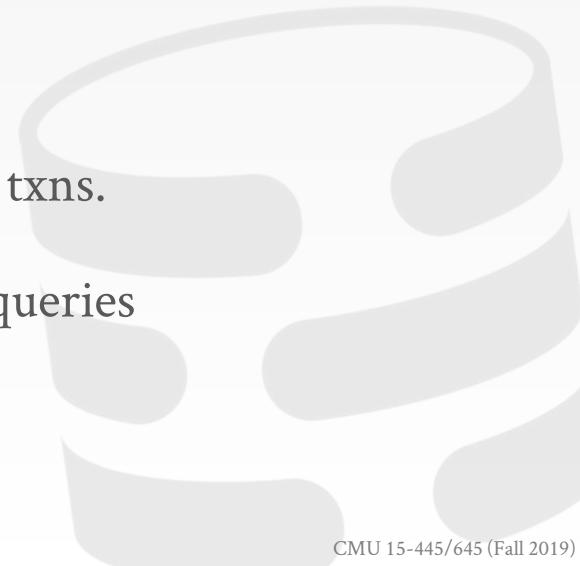
LOGGING SCHEMES

Physical Logging

- Record the changes made to a specific location in the database.
- Example: **git diff**

Logical Logging

- Record the high-level operations executed by txns.
- Not necessarily restricted to single page.
- Example: The **UPDATE**, **DELETE**, and **INSERT** queries invoked by a txn.



PHYSICAL VS. LOGICAL LOGGING

Logical logging requires less data written in each log record than physical logging.

Difficult to implement recovery with logical logging if you have concurrent txns.

- Hard to determine which parts of the database may have been modified by a query before crash.
- Also takes longer to recover because you must re-execute every txn all over again.

PHYSIOLOGICAL LOGGING

Hybrid approach where log records target a single page but do not specify data organization of the page.

This is the most popular approach.



LOGGING SCHEMES

```
UPDATE foo SET val = XYZ WHERE id = 1;
```

Physical

```
<T1,  
Table=X,  
Page=99,  
Offset=4,  
Before=ABC,  
After=XYZ>  
<T1,  
Index=X_PKEY,  
Page=45,  
Offset=9,  
Key=(1,Record1)>
```

Logical

```
<T1,  
Query="UPDATE foo  
SET val=XYZ  
WHERE id=1">
```

Physiological

```
<T1,  
Table=X,  
Page=99,  
ObjectId=1,  
Before=ABC,  
After=XYZ>  
<T1,  
Index=X_PKEY,  
IndexPage=45,  
Key=(1,Record1)>
```

CHECKPOINTS

The WAL will grow forever.

After a crash, the DBMS has to replay the entire log which will take a long time.

The DBMS periodically takes a checkpoint where it flushes all buffers out to disk.



CHECKPOINTS

Output onto stable storage all log records currently residing in main memory.

Output to the disk all modified blocks.

Write a <CHECKPOINT> entry to the log and flush to stable storage.

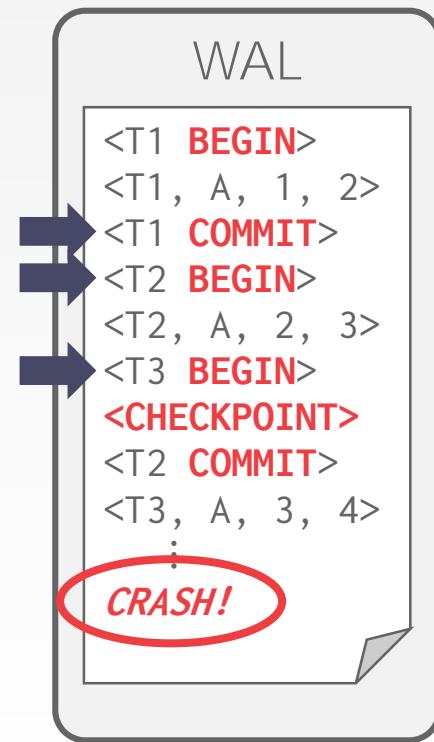


CHECKPOINTS

Any txn that committed before the checkpoint is ignored (T_1).

$T_2 + T_3$ did not commit before the last checkpoint.

- Need to redo T_2 because it committed after checkpoint.
- Need to undo T_3 because it did not commit before the crash.

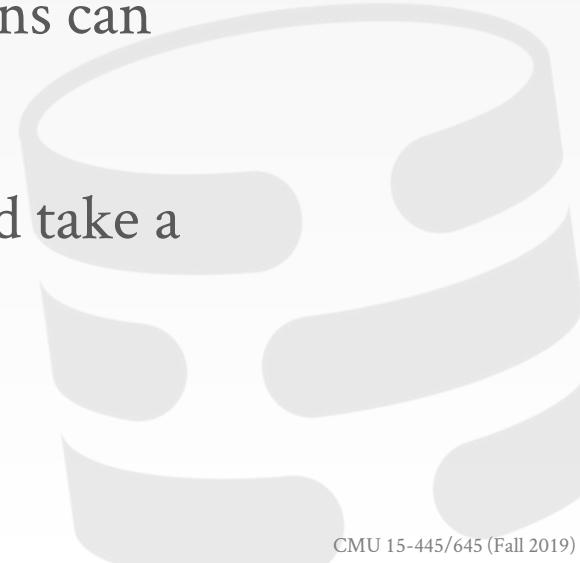


CHECKPOINTS – CHALLENGES

We have to stall all txns when take a checkpoint to ensure a consistent snapshot.

Scanning the log to find uncommitted txns can take a long time.

Not obvious how often the DBMS should take a checkpoint...



CHECKPOINTS – FREQUENCY

Checkpointing too often causes the runtime performance to degrade.

→ System spends too much time flushing buffers.

But waiting a long time is just as bad:

→ The checkpoint will be large and slow.
→ Makes recovery time much longer.



CONCLUSION

Write-Ahead Logging is (almost) always the best approach to handle loss of volatile storage.

- Use incremental updates (**STEAL** + **NO-FORCE**) with checkpoints.
- On recovery: undo uncommitted txns + redo committed txns.



NEXT CLASS

Recovery with ARIES.



21

Database Recovery



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

CRASH RECOVERY

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures.

Recovery algorithms have two parts:

- Actions during normal txn processing to ensure that the DBMS can recover from a failure.
- Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

Today

ARIES

Algorithms for Recovery and Isolation Exploiting Semantics

Developed at IBM Research in early 1990s for the DB2 DBMS.

Not all systems implement ARIES exactly as defined in this paper but they're close enough.

ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN
IBM Almaden Research Center
and

DON HADERLE
IBM Santa Teresa Laboratory
and

BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ
IBM Almaden Research Center

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log, it is ensured that rollbacks do those written during the transaction, a bounded number of logins is incurred. Logrollbacks are the focus of repeated failures during restart or of nested rollbacks. We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system. ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying levels of durability. By employing parallelism during both parallelized rollbacks and logical locks, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

Authors' addresses: C. Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 0362-5915/92/0300-0094 \$1.50

ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992, Pages 94-162

ARIES – MAIN IDEAS

Write-Ahead Logging:

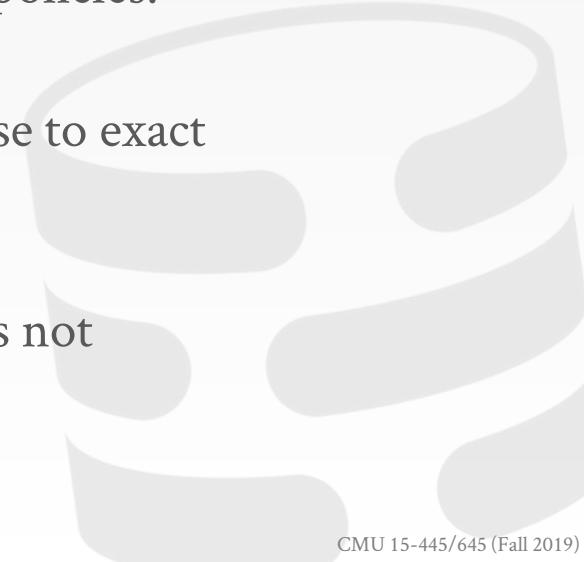
- Any change is recorded in log on stable storage before the database change is written to disk.
- Must use **STEAL** + **NO-FORCE** buffer pool policies.

Repeating History During Redo:

- On restart, retrace actions and restore database to exact state before crash.

Logging Changes During Undo:

- Record undo actions to log to ensure action is not repeated in the event of repeated failures.



TODAY'S AGENDA

Log Sequence Numbers

Normal Commit & Abort Operations

Fuzzy Checkpointing

Recovery Algorithm

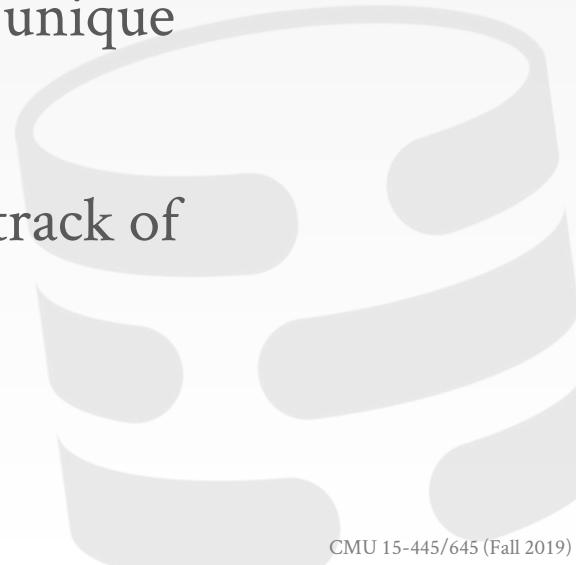


WAL RECORDS

We need to extend our log record format from last class to include additional info.

Every log record now includes a globally unique *log sequence number* (LSN).

Various components in the system keep track of *LSNs* that pertain to them...



LOG SEQUENCE NUMBERS

Name	Where	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	page _x	Newest update to page _x
recLSN	page _x	Oldest update to page _x since it was last flushed
lastLSN	T _i	Latest record of txn T _i
MasterRecord	Disk	LSN of latest checkpoint

WRITING LOG RECORDS

Each data page contains a **pageLSN**.

→ The *LSN* of the most recent update to that page.

System keeps track of **flushedLSN**.

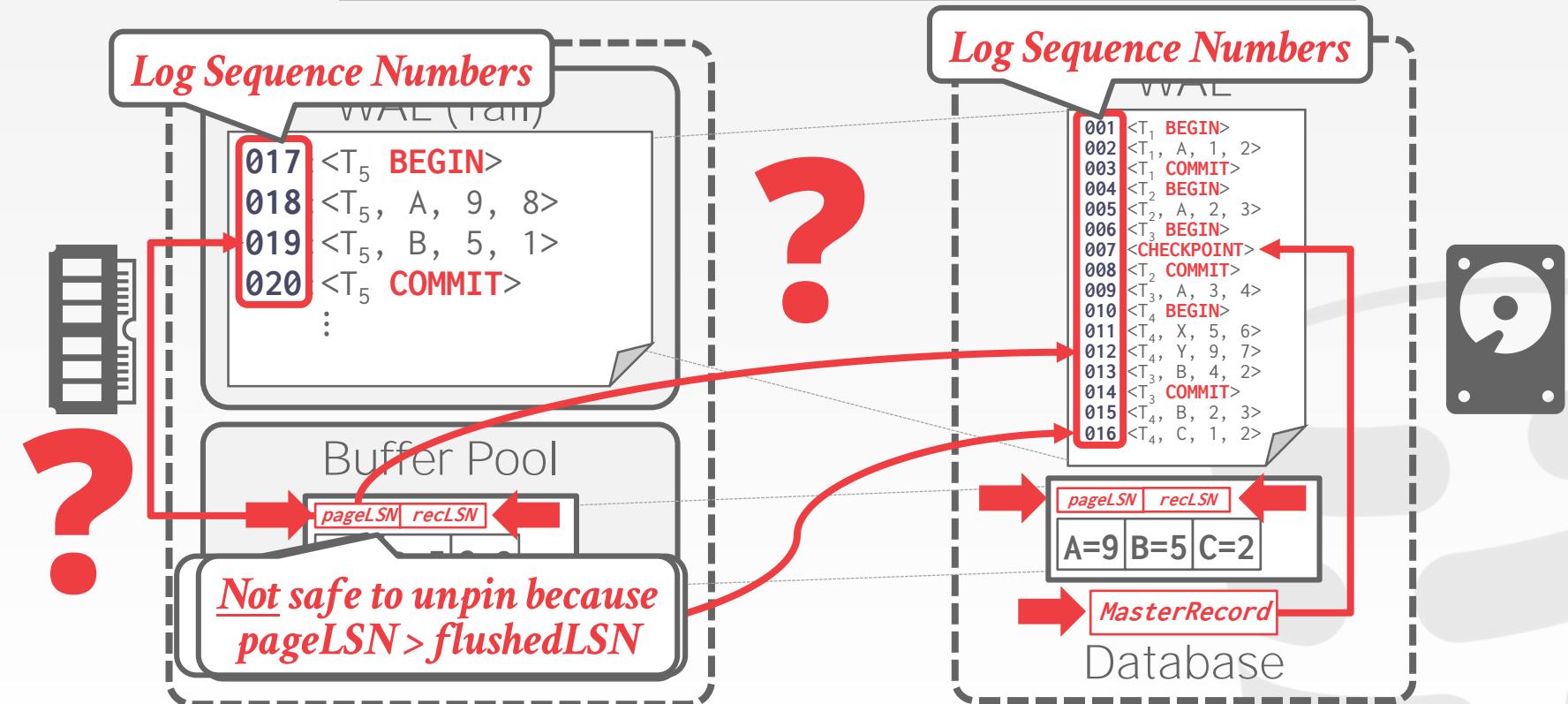
→ The max *LSN* flushed so far.

Before page **x** can be written to disk, we must flush log at least to the point where:

→ $\text{pageLSN}_x \leq \text{flushedLSN}$



WRITING LOG RECORDS



WRITING LOG RECORDS

All log records have an *LSN*.

Update the **pageLSN** every time a txn modifies a record in the page.

Update the **flushedLSN** in memory every time the DBMS writes out the WAL buffer to disk.

NORMAL EXECUTION

Each txn invokes a sequence of reads and writes, followed by commit or abort.

Assumptions in this lecture:

- All log records fit within a single page.
- Disk writes are atomic.
- Single-versioned tuples with Strict 2PL.
- **STEAL + NO-FORCE** buffer management with WAL.



TRANSACTION COMMIT

Write **COMMIT** record to log.

All log records up to txn's **COMMIT** record are flushed to disk.

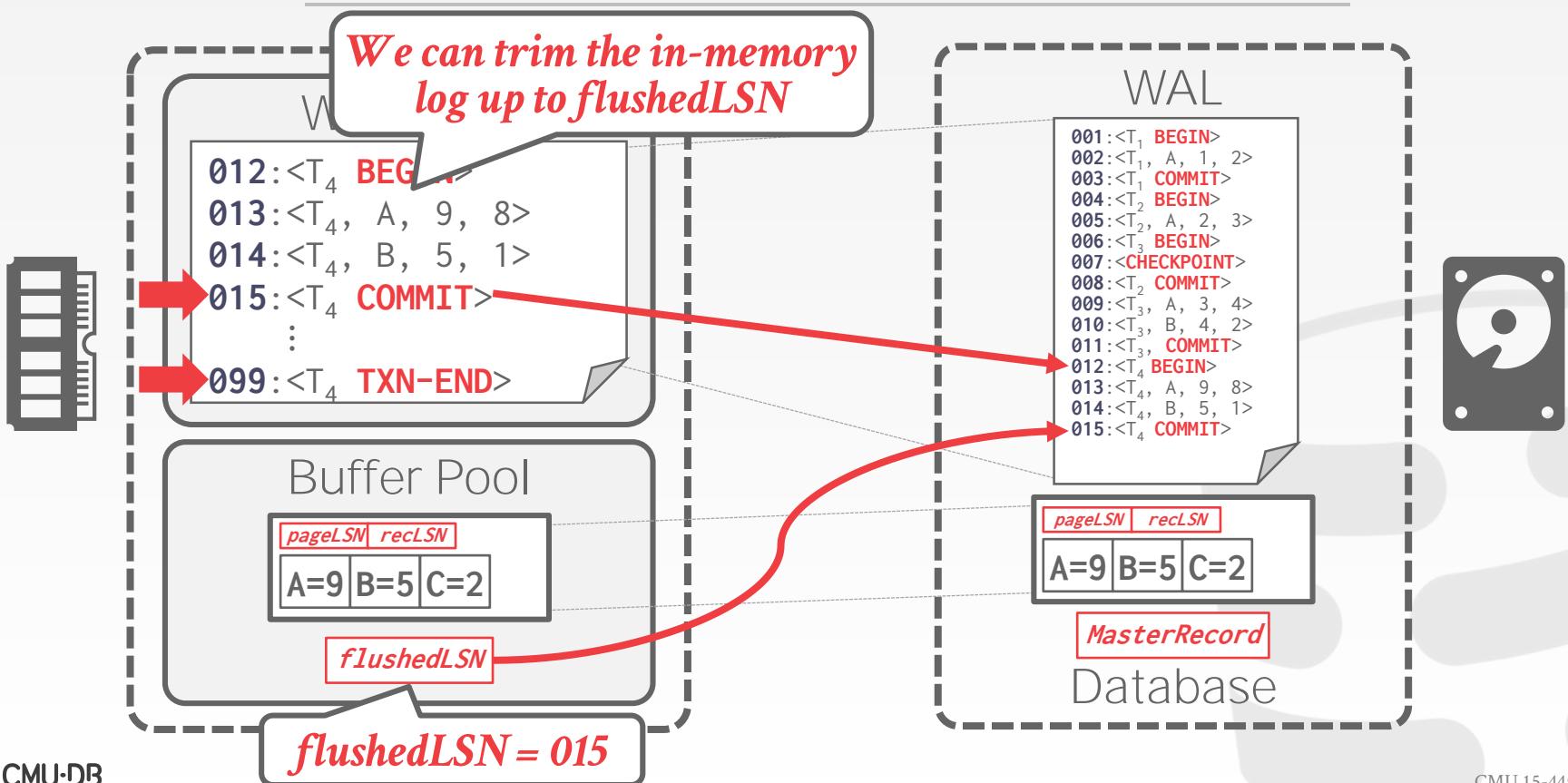
- Note that log flushes are sequential, synchronous writes to disk.
- Many log records per log page.

When the commit succeeds, write a special **TXN-END** record to log.

- This does not need to be flushed immediately.

TRANSACTION COMMIT

We can trim the in-memory log up to flushedLSN



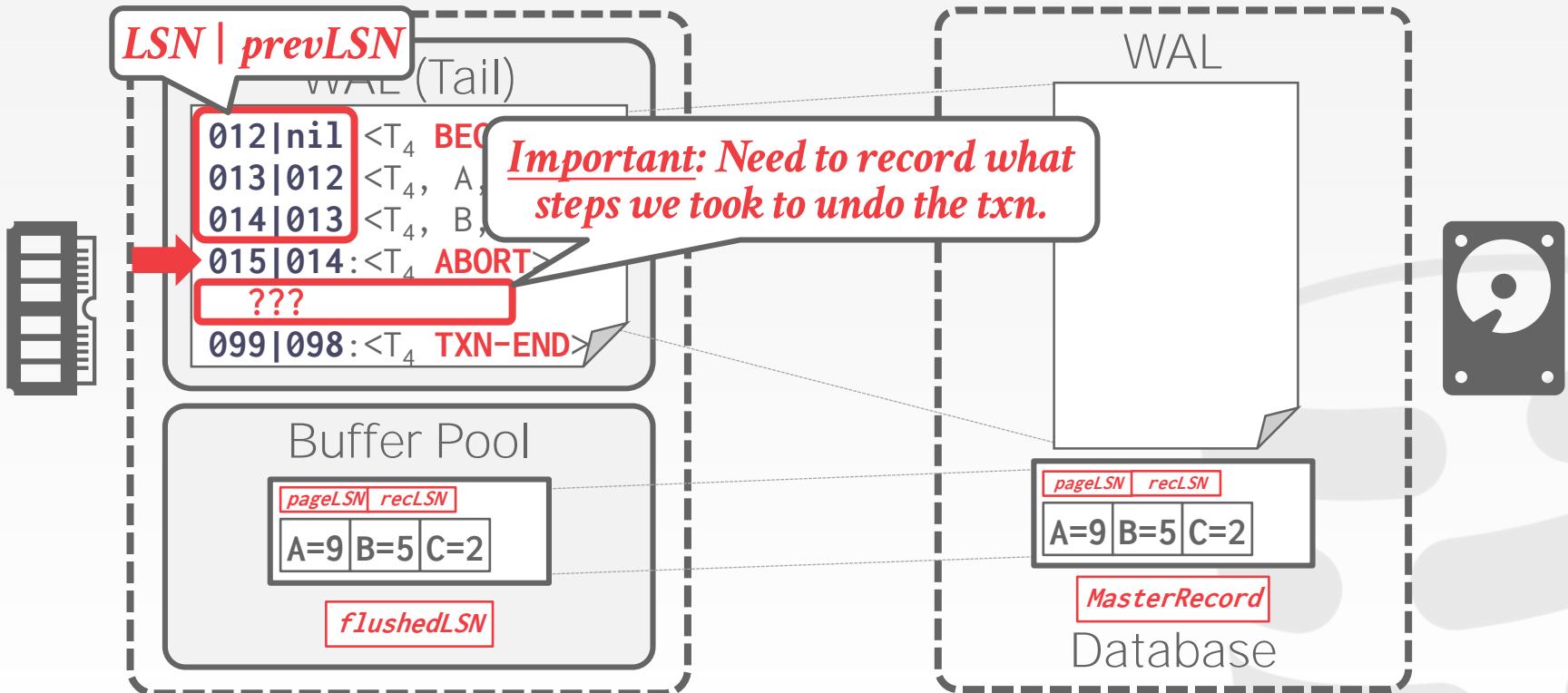
TRANSACTION ABORT

Aborting a txn is actually a special case of the ARIES undo operation applied to only one transaction.

We need to add another field to our log records:

- **prevLSN**: The previous *LSN* for the txn.
- This maintains a linked-list for each txn that makes it easy to walk through its records.

TRANSACTION ABORT



COMPENSATION LOG RECORDS

A **CLR** describes the actions taken to undo the actions of a previous update record.

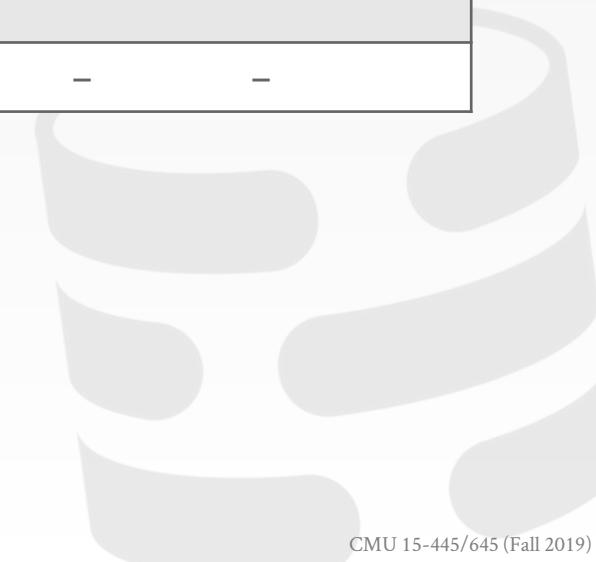
It has all the fields of an update log record plus the **undoNext** pointer (the next-to-be-undone LSN).

CLRs are added to log like any other record.

TRANSACTION ABORT – CLR EXAMPLE



LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNext
001	nil	T ₁	BEGIN	-	-	-	-
002	001	T ₁	UPDATE	A	30	40	-
⋮							
011	002	T ₁	ABORT	-	-	-	-



TRANSACTION ABORT – CLR EXAMPLE



LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNext
001	nil	T ₁	BEGIN	-	-	-	-
002	001	T ₁	UPDATE	A	30	40	-
⋮							
011	002	T ₁	ABORT	-	-	-	-
⋮							
026	011	T ₁	CLR-002	A	40	30	001

The LSN of the next log record to be undone.

TRANSACTION ABORT – CLR EXAMPLE



LSN	prevLSN	TxnId	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	-	-	-	-
002	001	T_1	UPDATE	A	30	40	-
⋮							
011	002	T_1	ABORT	-	-	-	-
⋮							
026	011	T_1	CLR-002	A	40	30	001
027	026	T_1	TXN-END	-	-	-	nil

ABORT ALGORITHM

First write an **ABORT** record to log for the txn.

Then play back the txn's updates in reverse order.

For each update record:

- Write a **CLR** entry to the log.
- Restore old value.

At end, write a **TXN-END** log record.

Notice: **CLRs** never need to be undone.



TODAY'S AGENDA

Log Sequence Numbers

~~Normal Commit & Abort Operations~~

Fuzzy Checkpointing

Recovery Algorithm



NON-FUZZY CHECKPOINTS

The DBMS halts everything when it takes a checkpoint to ensure a consistent snapshot:

- Halt the start of any new txns.
- Wait until all active txns finish executing.
- Flushes dirty pages on disk.

This is obviously bad...



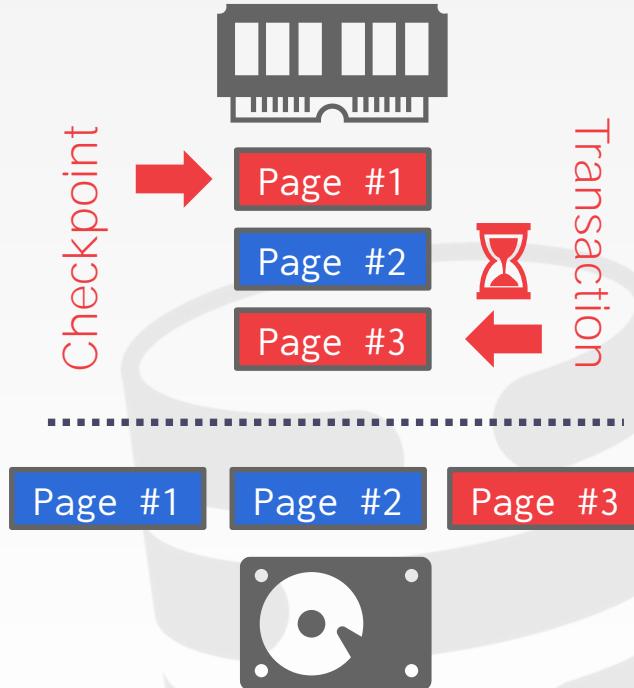
SLIGHTLY BETTER CHECKPOINTS

Pause modifying txns while the DBMS takes the checkpoint.

- Prevent queries from acquiring write latch on table/index pages.
- Don't have to wait until all txns finish before taking the checkpoint.

We must record internal state as of the beginning of the checkpoint.

- **Active Transaction Table (ATT)**
- **Dirty Page Table (DPT)**



ACTIVE TRANSACTION TABLE

One entry per currently active txn.

- **txnId**: Unique txn identifier.
- **status**: The current "mode" of the txn.
- **lastLSN**: Most recent *LSN* created by txn.

Entry removed when txn commits or aborts.

Txn Status Codes:

- **R** → Running
- **C** → Committing
- **U** → Candidate for Undo



DIRTY PAGE TABLE

Keep track of which pages in the buffer pool contain changes from uncommitted transactions.

One entry per dirty page in the buffer pool:

→ **recLSN**: The *LSN* of the log record that first caused the page to be dirty.

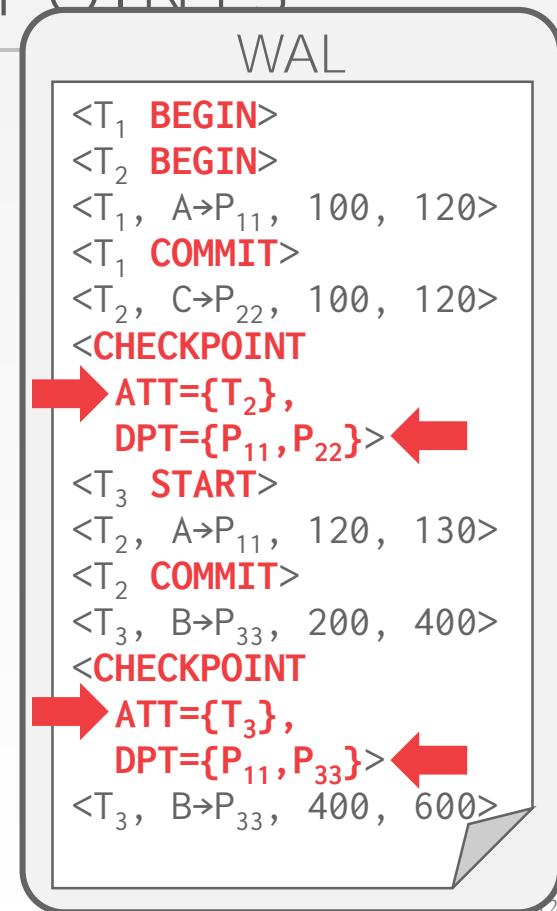


SLIGHTLY BETTER CHECKPOINTS

At the first checkpoint, T_2 is still running and there are two dirty pages (P_{11}, P_{22}).

At the second checkpoint, T_3 is active and there are two dirty pages (P_{11}, P_{33}).

This still is not ideal because the DBMS must stall txns during checkpoint...



FUZZY CHECKPOINTS

A *fuzzy checkpoint* is where the DBMS allows active txns to continue the run while the system flushes dirty pages to disk.

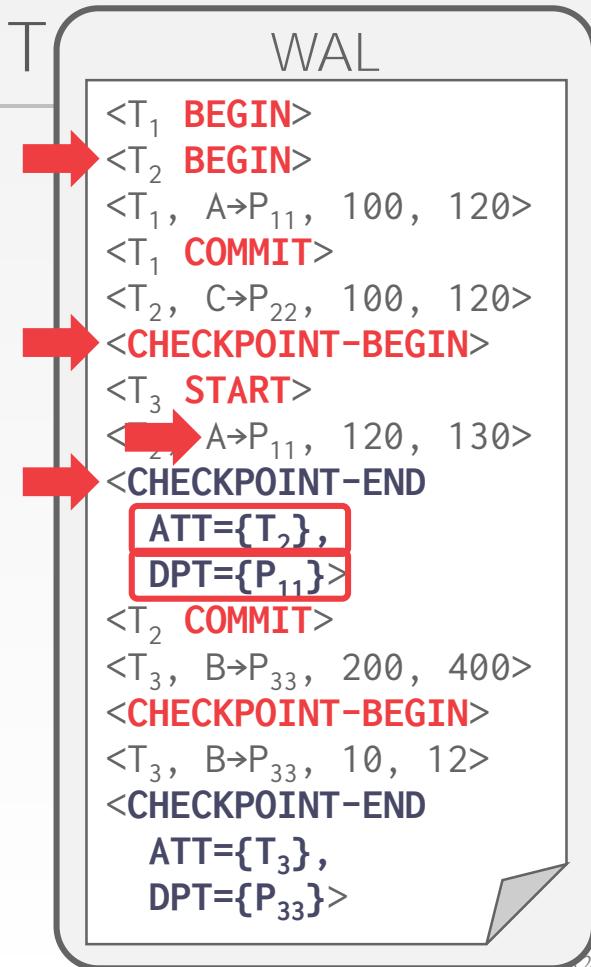
New log records to track checkpoint boundaries:

- **CHECKPOINT-BEGIN**: Indicates start of checkpoint
- **CHECKPOINT-END**: Contains ATT + DPT.

FUZZY CHECKPOINT

The *LSN* of the **CHECKPOINT-BEGIN** record is written to the database's **MasterRecord** entry on disk when the checkpoint successfully completes.

Any txn that starts after the checkpoint is excluded from the ATT in the **CHECKPOINT-END** record.



ARIES – RECOVERY PHASES

Phase #1 – Analysis

- Read WAL from last checkpoint to identify dirty pages in the buffer pool and active txns at the time of the crash.

Phase #2 – Redo

- Repeat all actions starting from an appropriate point in the log (even txns that will abort).

Phase #3 – Undo

- Reverse the actions of txns that did not commit before the crash.

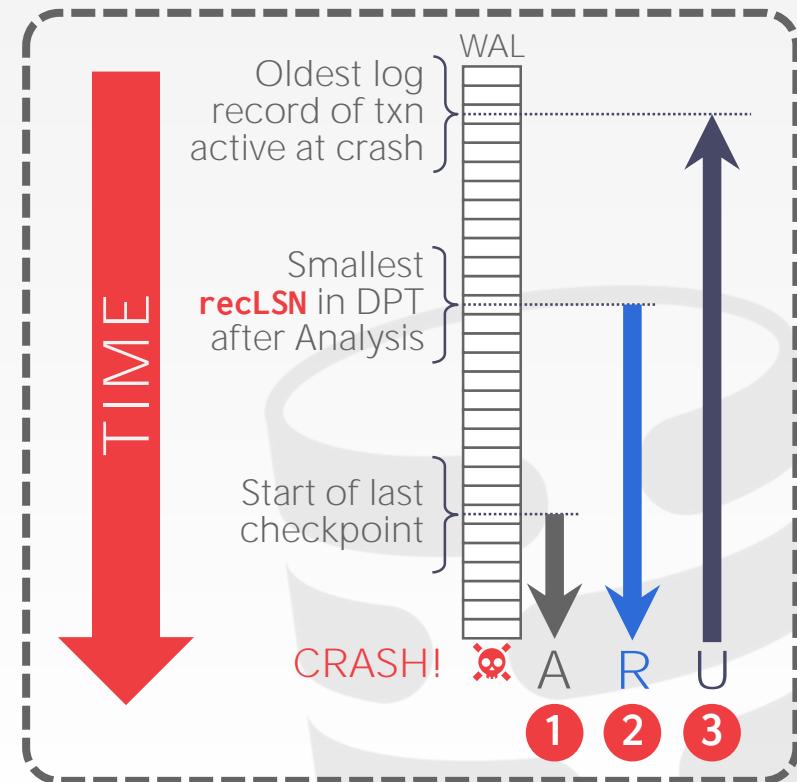
ARIES – OVERVIEW

Start from last **BEGIN-CHECKPOINT**
found via **MasterRecord**.

Analysis: Figure out which txns
committed or failed since checkpoint.

Redo: Repeat all actions.

Undo: Reverse effects of failed txns.



ANALYSIS PHASE

Scan log forward from last successful checkpoint.

If you find a **TXN-END** record, remove its corresponding txn from ATT.

All other records:

- Add txn to ATT with status **UNDO**.
- On commit, change txn status to **COMMIT**.

For **UPDATE** records:

- If page **P** not in DPT, add **P** to DPT, set its **recLSN=LSN**.



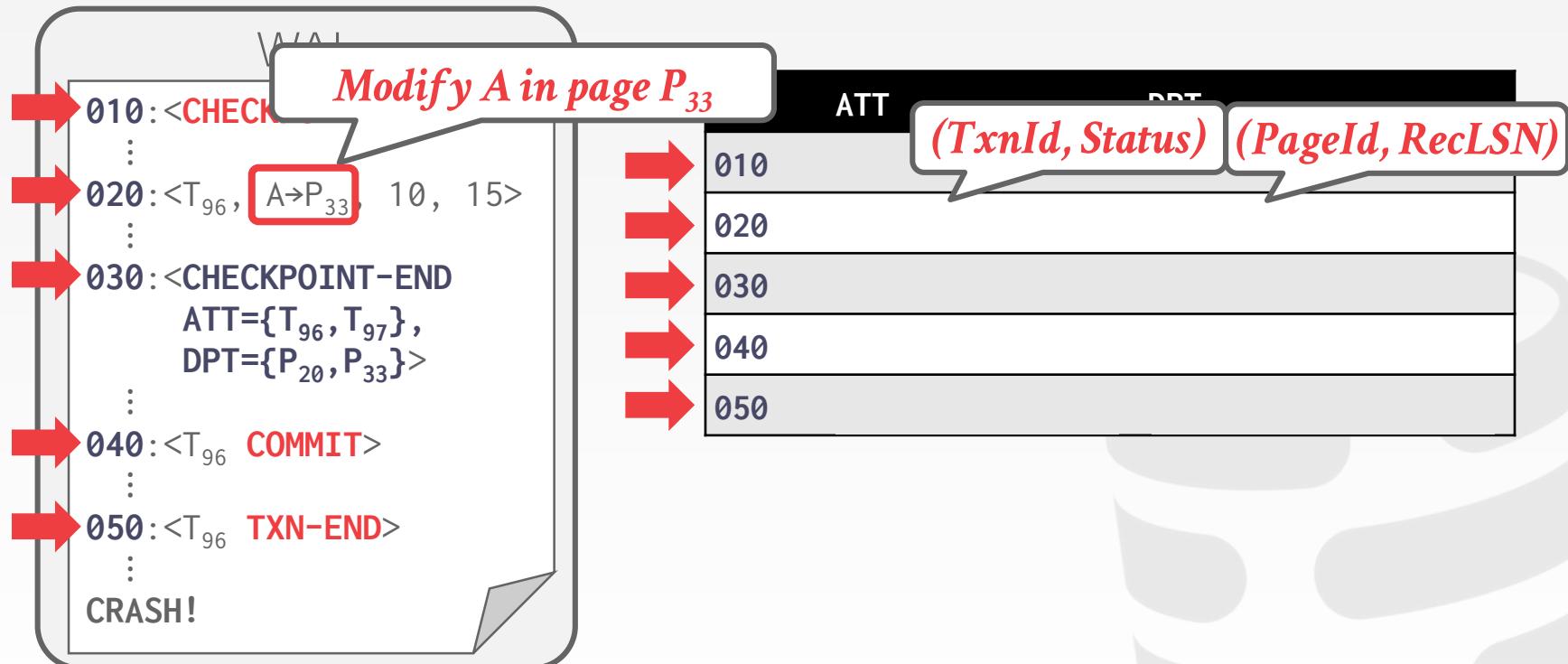
ANALYSIS PHASE

At end of the Analysis Phase:

- **ATT** tells the DBMS which txns were active at time of crash.
- **DPT** tells the DBMS which dirty pages might not have made it to disk.



ANALYSIS PHASE EXAMPLE



REDO PHASE

The goal is to repeat history to reconstruct state at the moment of the crash:

→ Reapply all updates (even aborted txns!) and redo **CLRs**.

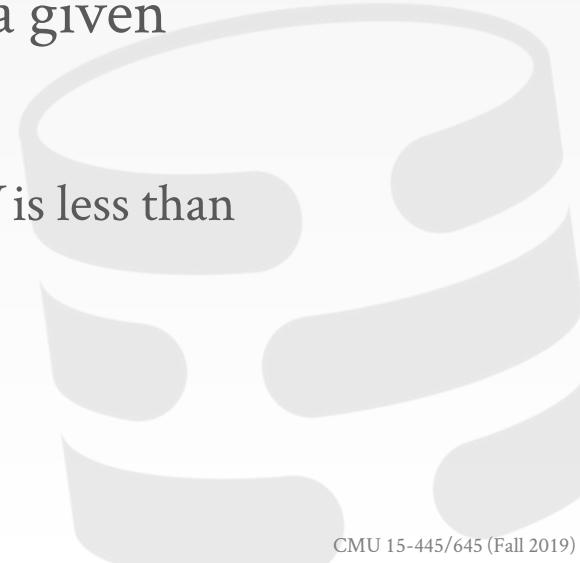
There techniques that allow the DBMS to avoid unnecessary reads/writes, but we will ignore that in this lecture...

REDO PHASE

Scan forward from the log record containing smallest **recLSN** in DPT.

For each update log record or **CLR** with a given **LSN**, redo the action unless:

- Affected page is not in DPT, or
- Affected page is in DPT but that record's **LSN** is less than the page's **recLSN**.



REDO PHASE

To redo an action:

- Reapply logged action.
- Set **pageLSN** to log record's **LSN**.
- No additional logging, no forced flushes!

At the end of Redo Phase, write **TXN-END** log records for all txns with status **C** and remove them from the ATT.



UNDO PHASE

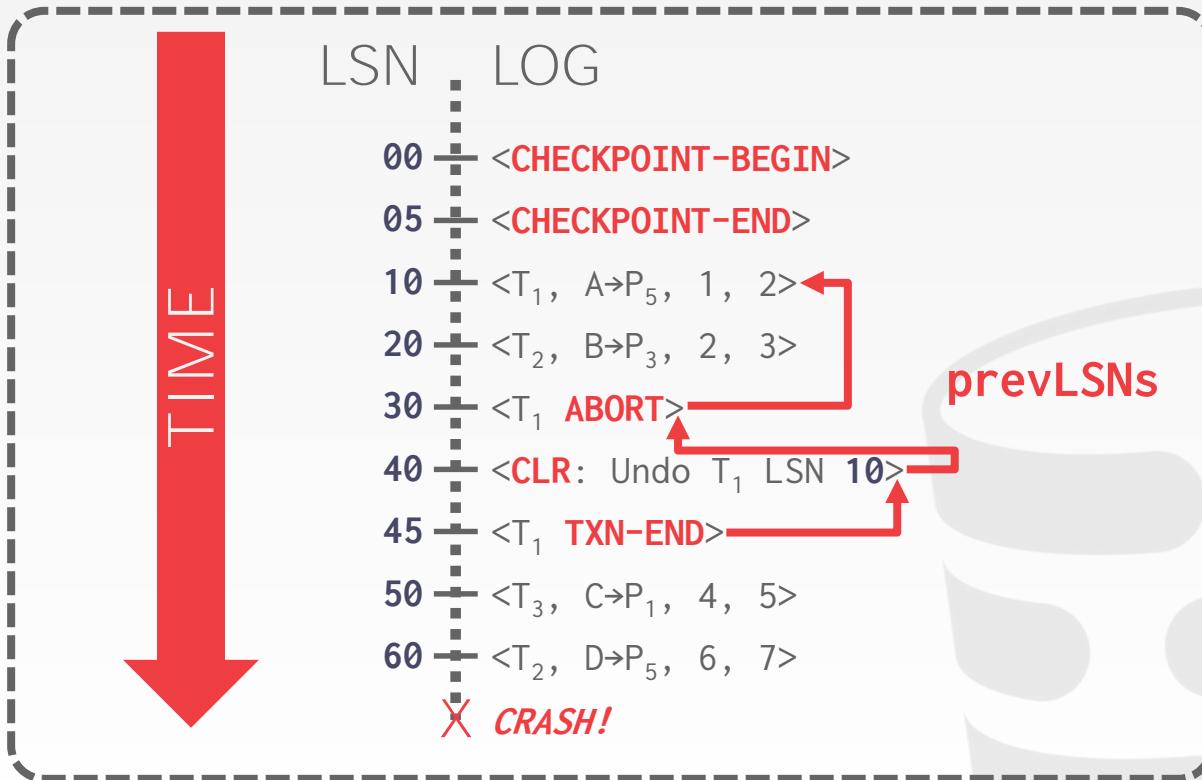
Undo all txns that were active at the time of crash and therefore will never commit.

→ These are all the txns with **U** status in the ATT after the Analysis Phase.

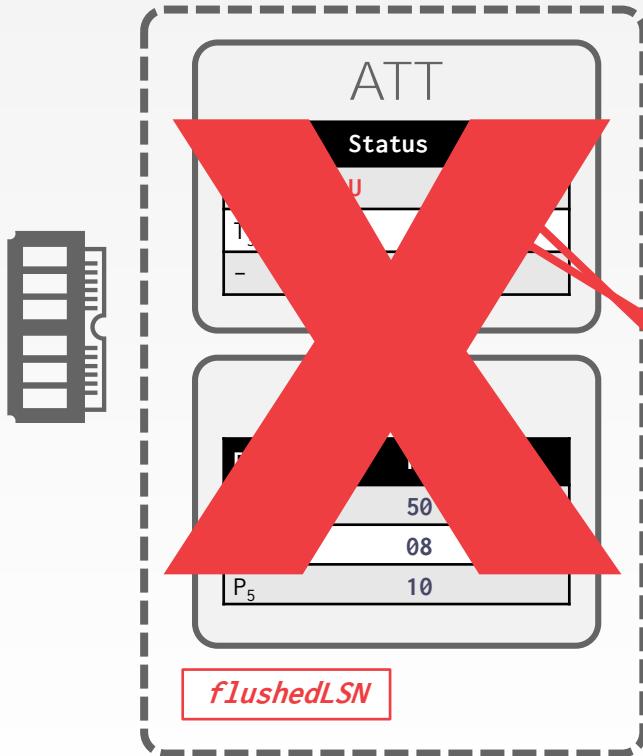
Process them in reverse **LSN** order using the **lastLSN** to speed up traversal.

Write a **CLR** for every modification.

FULL EXAMPLE



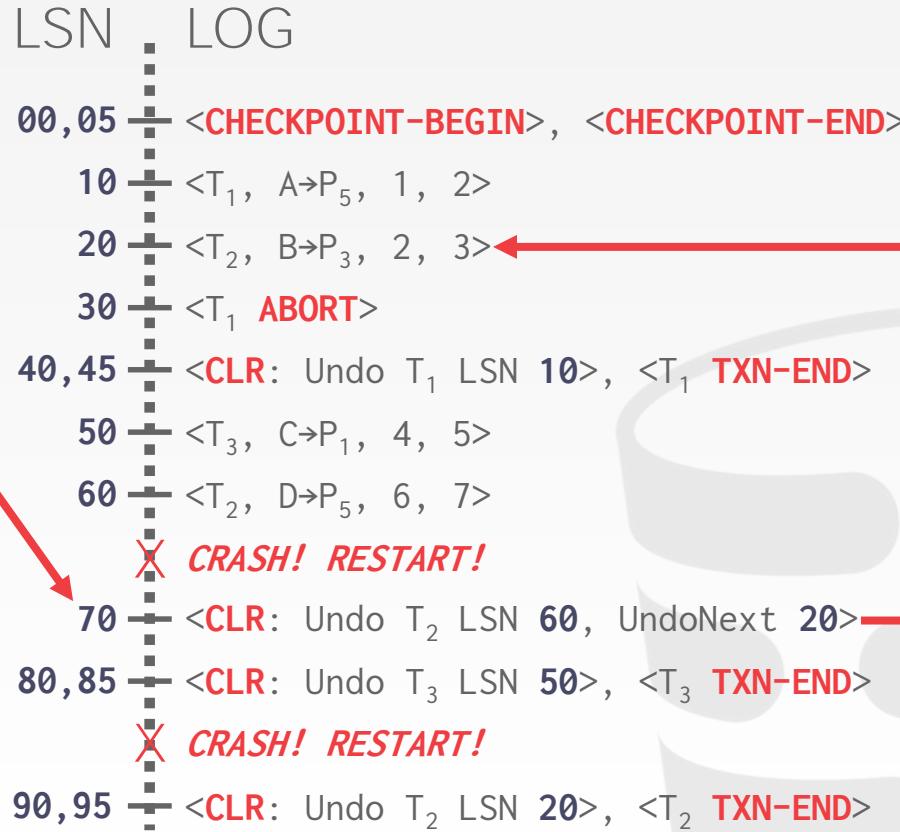
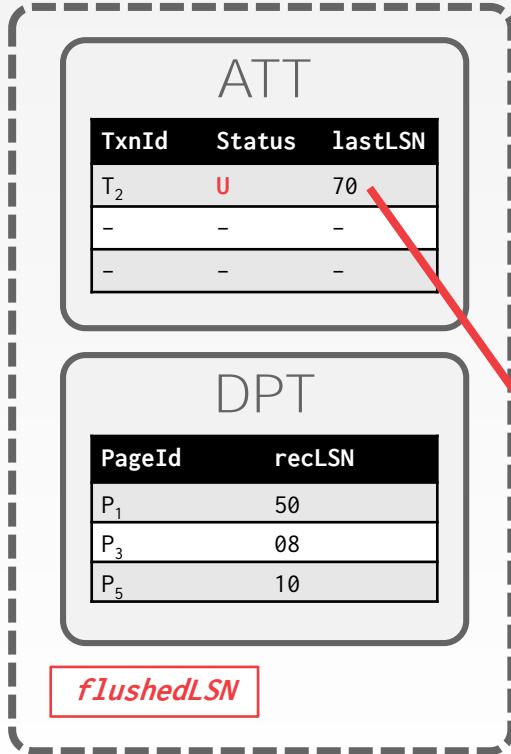
FULL EXAMPLE



LSN	LOG
00, 05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T ₁ , A→P ₅ , 1, 2>
20	<T ₂ , B→P ₃ , 2, 3>
30	<T ₁ ABORT>
40, 45	<CLR: Undo T ₁ LSN 10>, <T ₁ TXN-END>
50	<T ₃ , C→P ₁ , 4, 5>
60	<T ₂ , D→P ₅ , 6, 7>
70	X CRASH! RESTART!
80, 85	<CLR: Undo T ₂ LSN 60, UndoNext> <CLR: Undo T ₃ LSN 50>, <T ₃ TXN-END>
	X CRASH! RESTART!

*Flush dirty pages +
WAL to disk!*

FULL EXAMPLE



ADDITIONAL CRASH ISSUES (1)

What does the DBMS do if it crashes during recovery in the Analysis Phase?

- Nothing. Just run recovery again.

What does the DBMS do if it crashes during recovery in the Redo Phase?

- Again nothing. Redo everything again.



ADDITIONAL CRASH ISSUES (2)

How can the DBMS improve performance during recovery in the Redo Phase?

- Assume that it is not going to crash again and flush all changes to disk asynchronously in the background.

How can the DBMS improve performance during recovery in the Undo Phase?

- Lazily rollback changes before new txns access pages.
- Rewrite the application to avoid long-running txns.

CONCLUSION

Mains ideas of ARIES:

- WAL with **STEAL/NO-FORCE**
- Fuzzy Checkpoints (snapshot of dirty page ids)
- Redo everything since the earliest dirty page
- Undo txns that never commit
- Write **CLRs** when undoing, to survive failures during restarts

Log Sequence Numbers:

- **LSNs** identify log records; linked into backwards chains per transaction via **prevLSN**.
- **pageLSN** allows comparison of data page and log records.

NEXT CLASS

You now know how to build a single-node DBMS.

So now we can talk about distributed databases!



22

Introduction to Distributed Databases



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #5: Monday Dec 3rd @ 11:59pm

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 10th @ 11:59pm

Final Exam: Monday Dec 9th @ 5:30pm

ADMINISTRIVIA

Monday Dec 2th – Oracle Lecture

→ Shasank Chavan (VP In-Memory Databases)



Wednesday Dec 4th – Potpourri + Review

→ Vote for what system you want me to talk about.

→ <https://cmudb.io/f19-systems>



Sunday Nov 24th – Extra Credit Check

→ Submit your extra credit assignment early to get feedback from me.

UPCOMING DATABASE EVENTS

Oracle Research Talk

- Tuesday December 4th @ 12:00pm
- CIC 4th Floor

ORACLE®



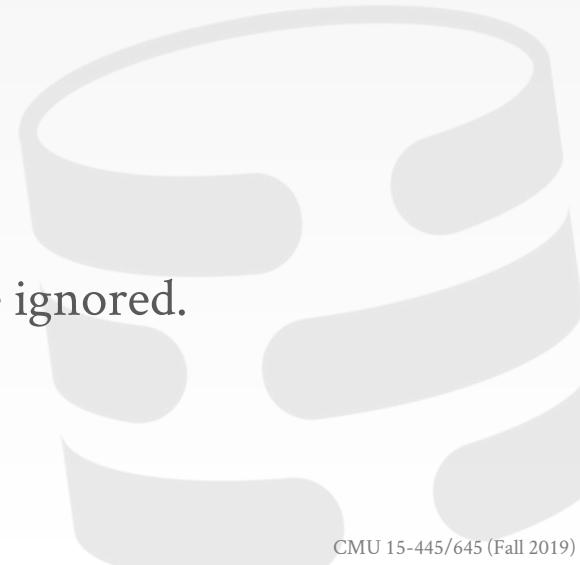
PARALLEL VS. DISTRIBUTED

Parallel DBMSs:

- Nodes are physically close to each other.
- Nodes connected with high-speed LAN.
- Communication cost is assumed to be small.

Distributed DBMSs:

- Nodes can be far from each other.
- Nodes connected using public network.
- Communication cost and problems cannot be ignored.



DISTRIBUTED DBMSs

Use the building blocks that we covered in single-node DBMSs to now support transaction processing and query execution in distributed environments.

- Optimization & Planning
- Concurrency Control
- Logging & Recovery



TODAY'S AGENDA

System Architectures

Design Issues

Partitioning Schemes

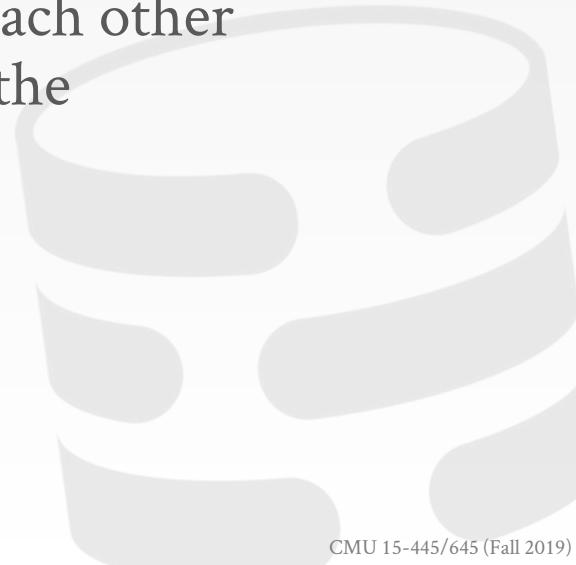
Distributed Concurrency Control



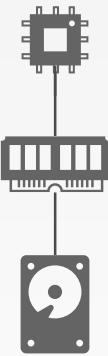
SYSTEM ARCHITECTURE

A DBMS's system architecture specifies what shared resources are directly accessible to CPUs.

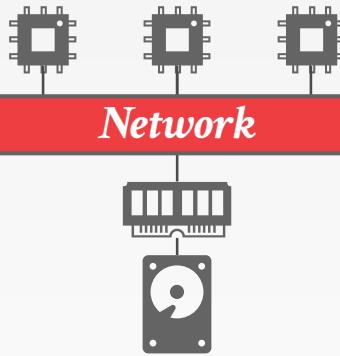
This affects how CPUs coordinate with each other and where they retrieve/store objects in the database.



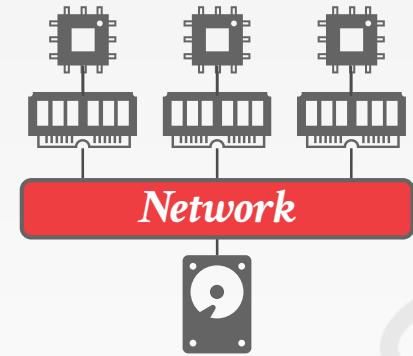
SYSTEM ARCHITECTURE



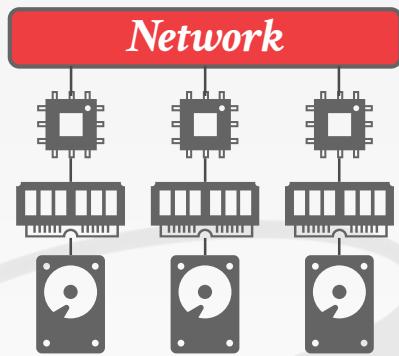
Shared
Everything



Shared
Memory



Shared
Disk

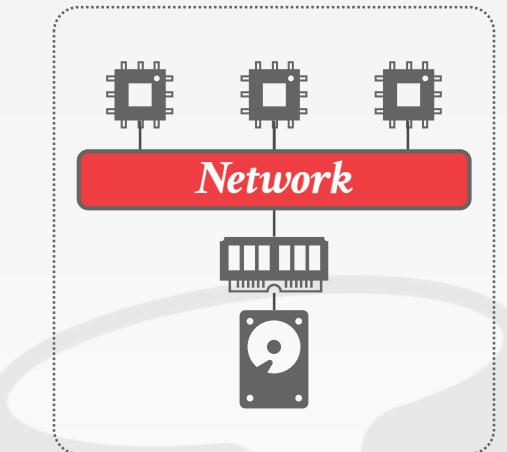


Shared
Nothing

SHARED MEMORY

CPUs have access to common memory address space via a fast interconnect.

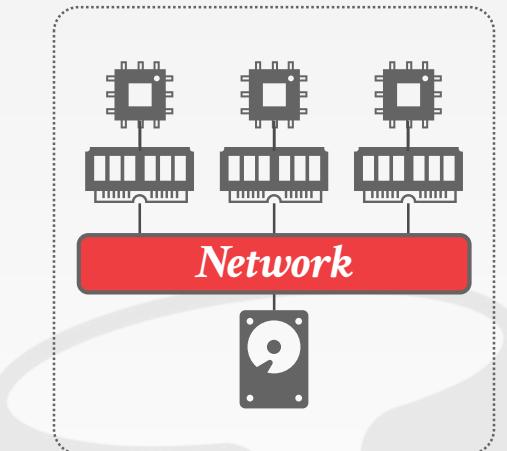
- Each processor has a global view of all the in-memory data structures.
- Each DBMS instance on a processor has to "know" about the other instances.



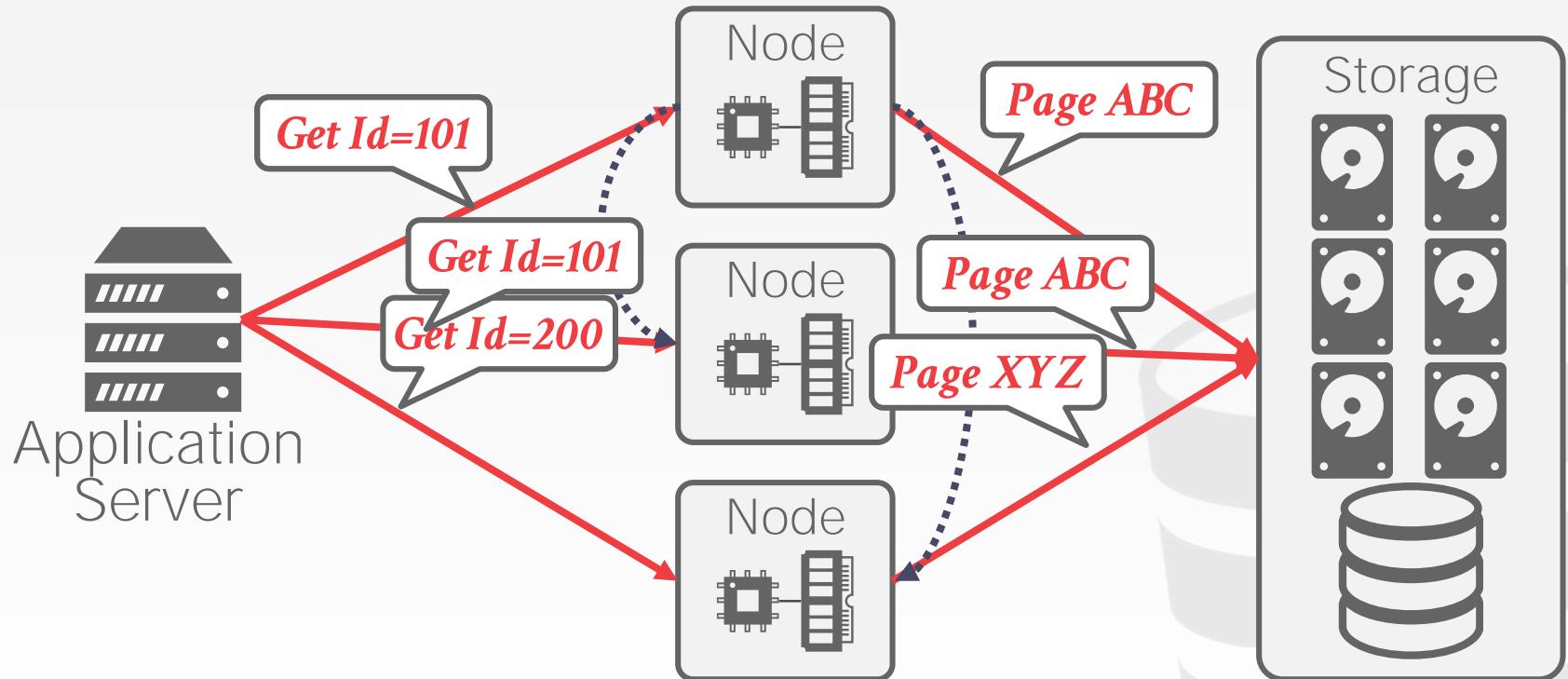
SHARED DISK

All CPUs can access a single logical disk directly via an interconnect, but each have their own private memories.

- Can scale execution layer independently from the storage layer.
- Must send messages between CPUs to learn about their current state.



SHARED DISK EXAMPLE

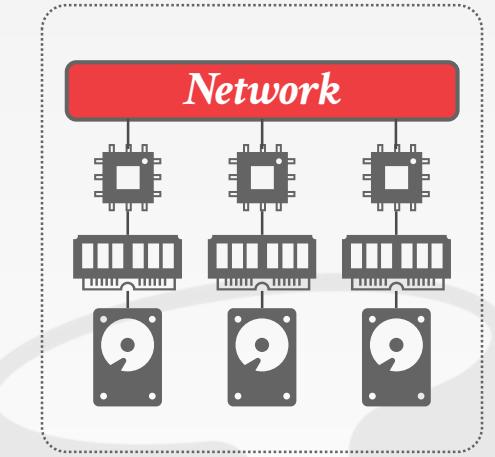


SHARED NOTHING

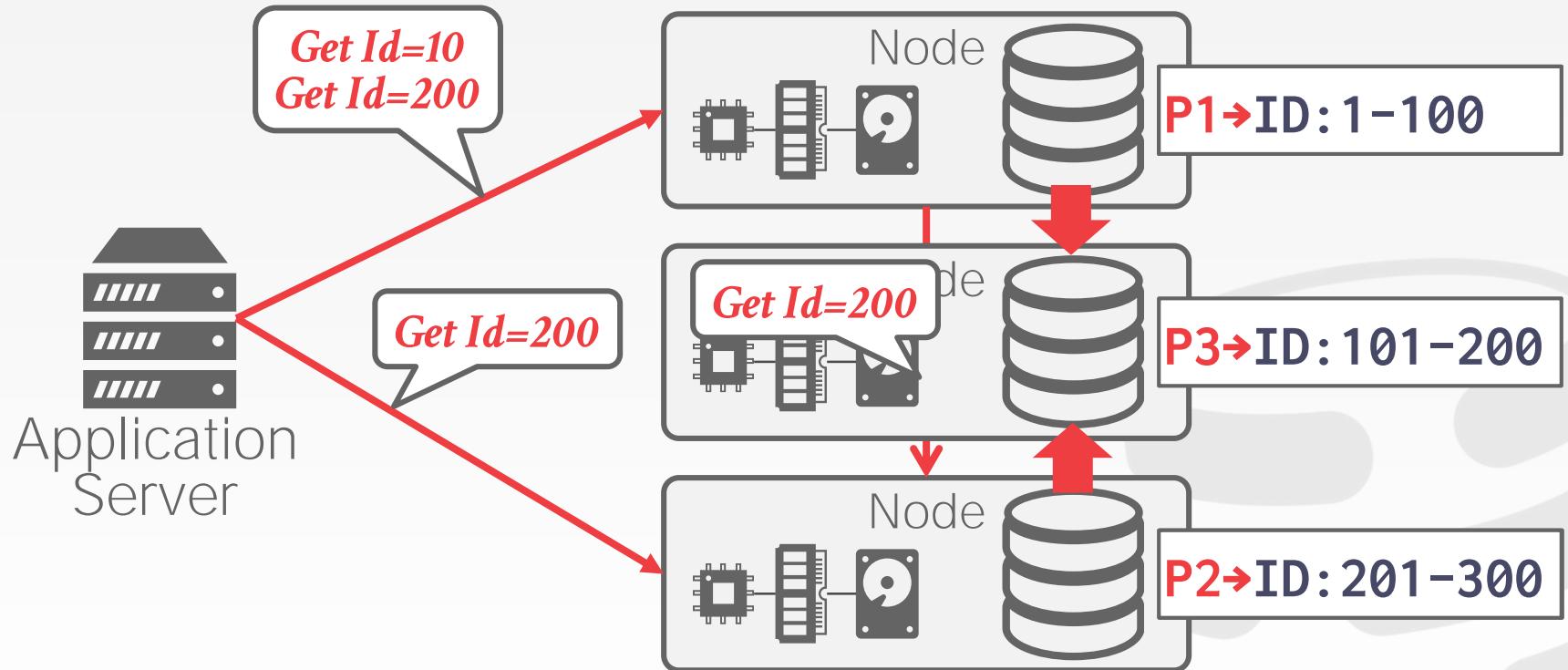
Each DBMS instance has its own CPU, memory, and disk.

Nodes only communicate with each other via network.

- Hard to increase capacity.
- Hard to ensure consistency.
- Better performance & efficiency.



SHARED NOTHING EXAMPLE



EARLY DISTRIBUTED DATABASE SYSTEMS

MUFFIN – UC Berkeley (1979)

SDD-1 – CCA (1979)

System R* – IBM Research (1984)

Gamma – Univ. of Wisconsin (1986)

NonStop SQL – Tandem (1987)



Stonebraker



Bernstein



Mohan



DeWitt



Gray

DESIGN ISSUES

How does the application find data?

How to execute queries on distributed data?

- Push query to data.
- Pull data to query.

How does the DBMS ensure correctness?



HOMOGENOUS VS. HETEROGENOUS

Approach #1: Homogenous Nodes

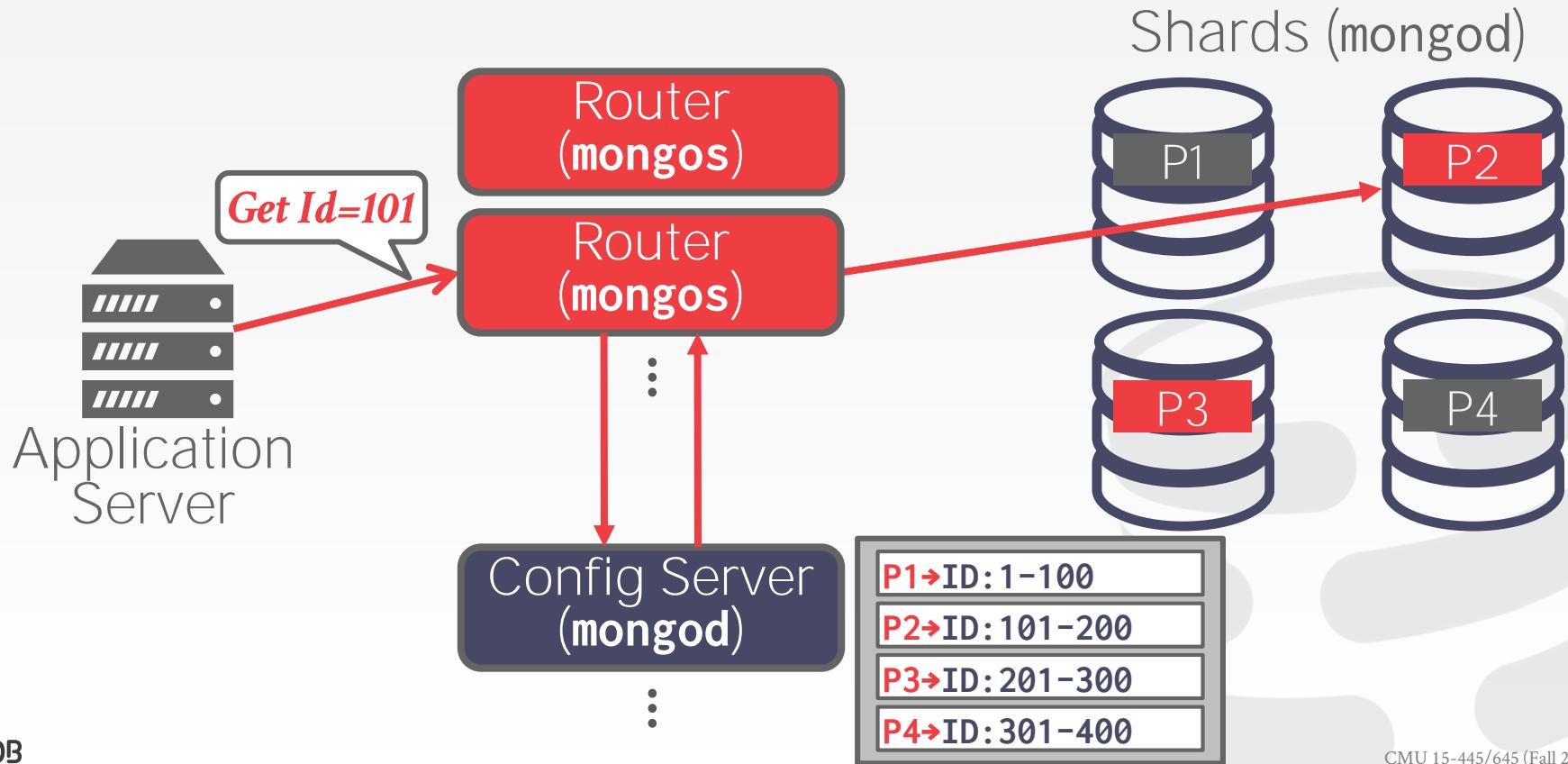
- Every node in the cluster can perform the same set of tasks (albeit on potentially different partitions of data).
- Makes provisioning and failover "easier".

Approach #2: Heterogenous Nodes

- Nodes are assigned specific tasks.
- Can allow a single physical node to host multiple "virtual" node types for dedicated tasks.



MONGODB HETEROGENOUS ARCHITECTURE



DATA TRANSPARENCY

Users should not be required to know where data is physically located, how tables are partitioned or replicated.

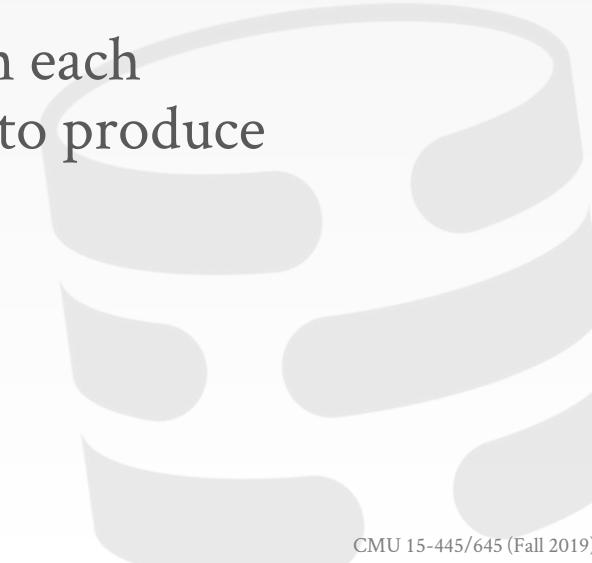
A SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

DATABASE PARTITIONING

Split database across multiple resources:

- Disks, nodes, processors.
- Sometimes called "sharding"

The DBMS executes query fragments on each partition and then combines the results to produce a single answer.



NAÏVE TABLE PARTITIONING

Each node stores one and only table.

Assumes that each node has enough storage space for a table.



NAÏVE TABLE PARTITIONING

Table1

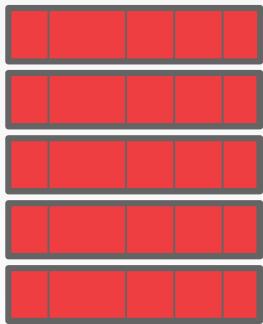
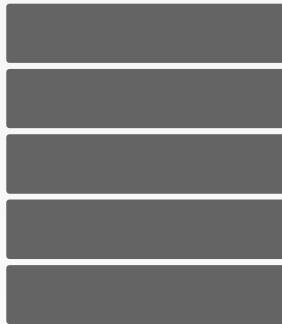
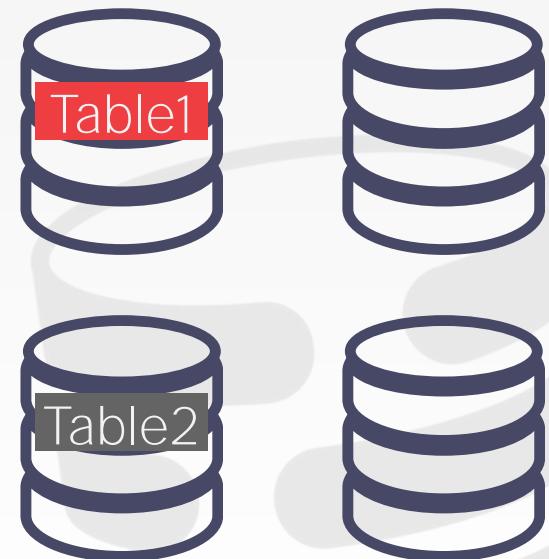


Table2



Partitions



Ideal Query:

```
SELECT * FROM table
```

HORIZONTAL PARTITIONING

Split a table's tuples into disjoint subsets.

- Choose column(s) that divides the database equally in terms of size, load, or usage.
- Hash Partitioning, Range Partitioning

The DBMS can partition a database **physical** (shared nothing) or **logically** (shared disk).

HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2019-11-29
102	b	YYY	2019-11-28
103	c	XYZ	2019-11-29
104	d	XYX	2019-11-27
105	e	XYY	2019-11-29

$\text{hash}(a) \% 4 = P2$

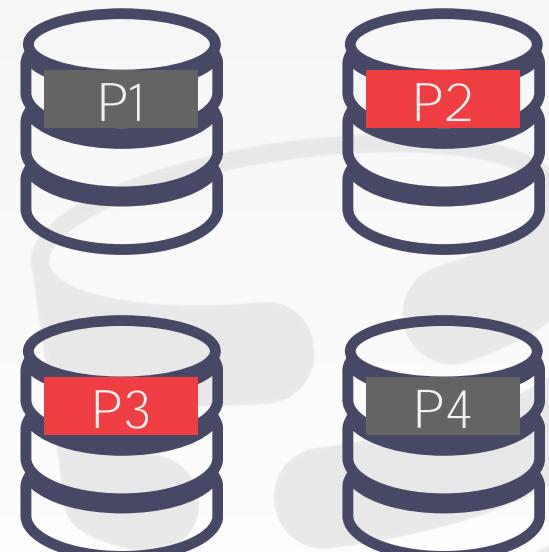
$\text{hash}(b) \% 4 = P4$

$\text{hash}(c) \% 4 = P3$

$\text{hash}(d) \% 4 = P2$

$\text{hash}(e) \% 4 = P1$

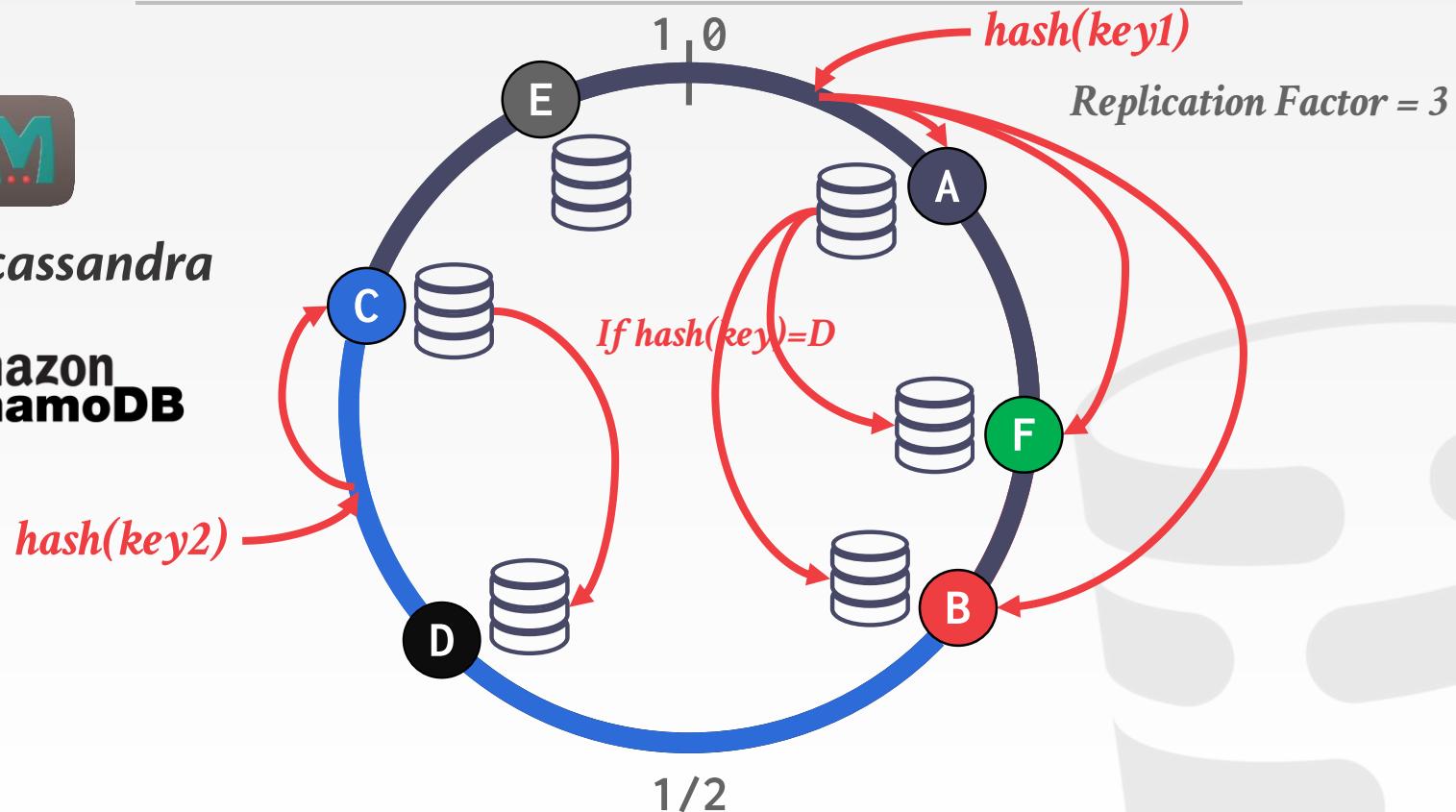
Partitions



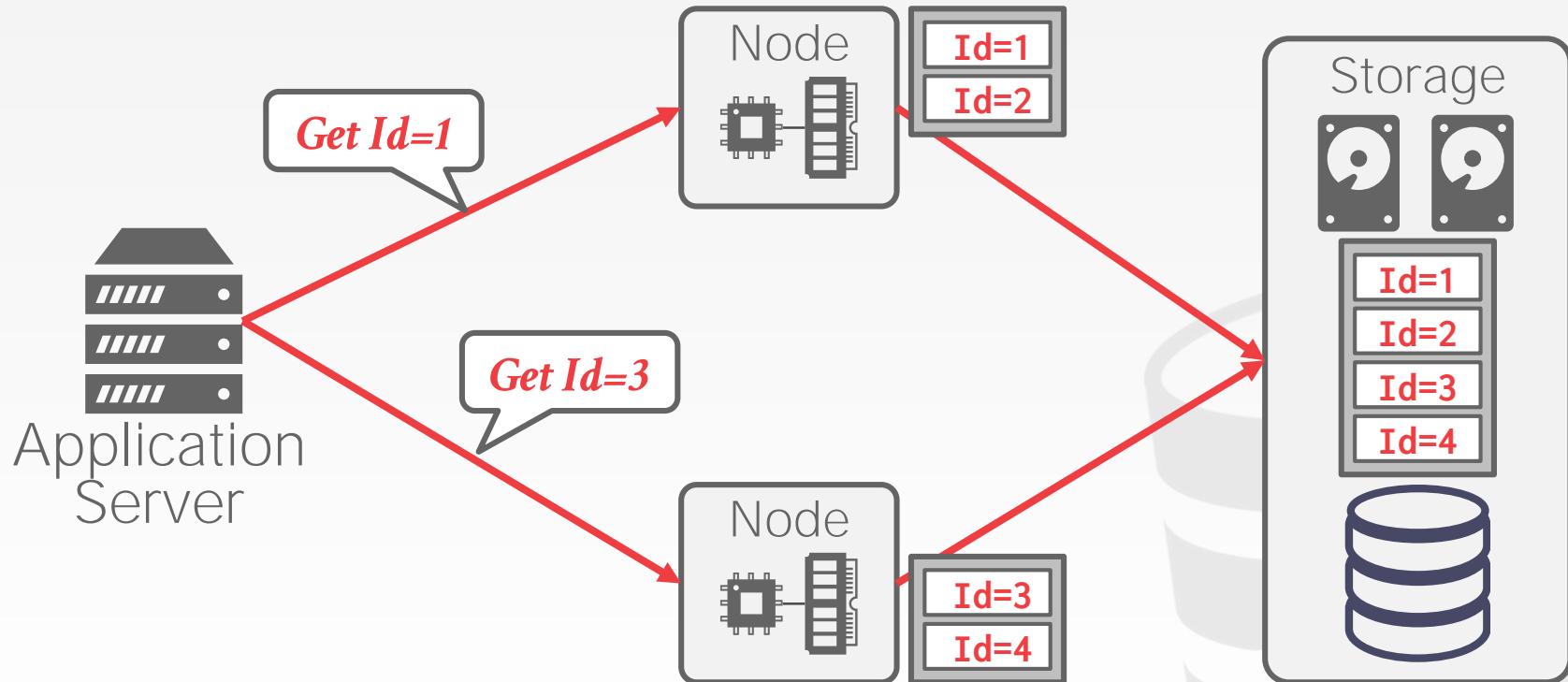
Ideal Query:

```
SELECT * FROM table
WHERE partitionKey = ?
```

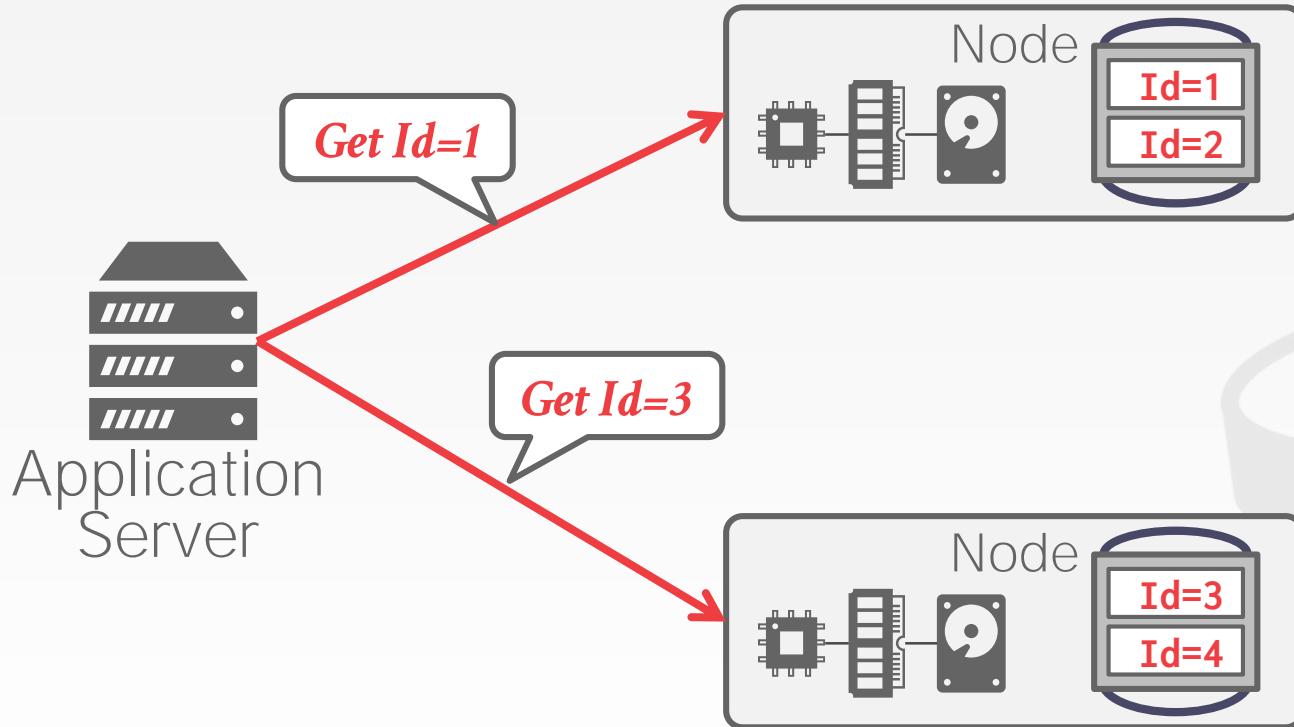
CONSISTENT HASHING



LOGICAL PARTITIONING



PHYSICAL PARTITIONING



SINGLE-NODE VS. DISTRIBUTED

A **single-node** txn only accesses data that is contained on one partition.

- The DBMS does not need coordinate the behavior concurrent txns running on other nodes.

A **distributed** txn accesses data at one or more partitions.

- Requires expensive coordination.



TRANSACTION COORDINATION

If our DBMS supports multi-operation and distributed txns, we need a way to coordinate their execution in the system.

Two different approaches:

- **Centralized**: Global "traffic cop".
- **Decentralized**: Nodes organize themselves.



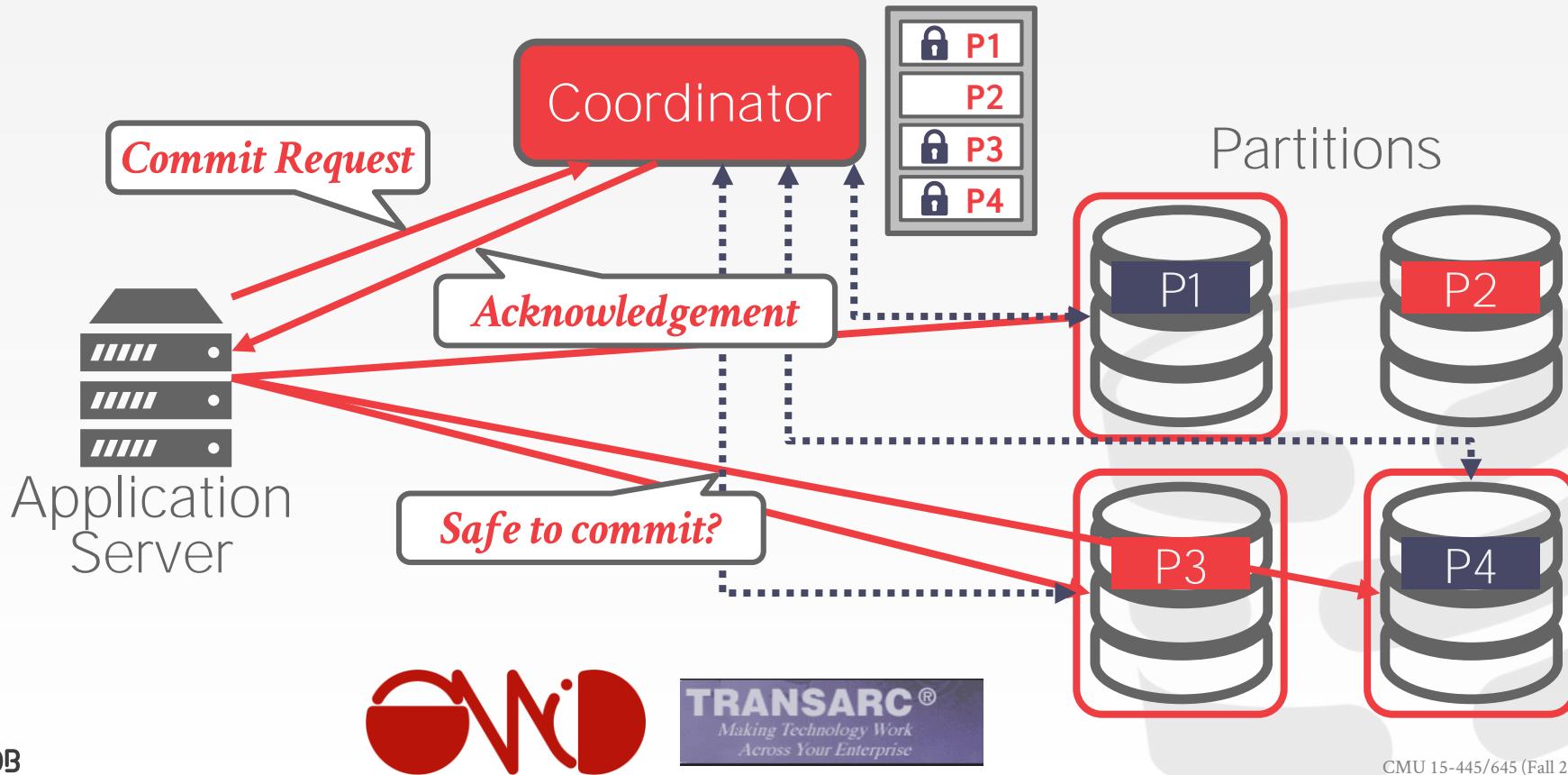
TP MONITORS

Example of a centralized coordinator.

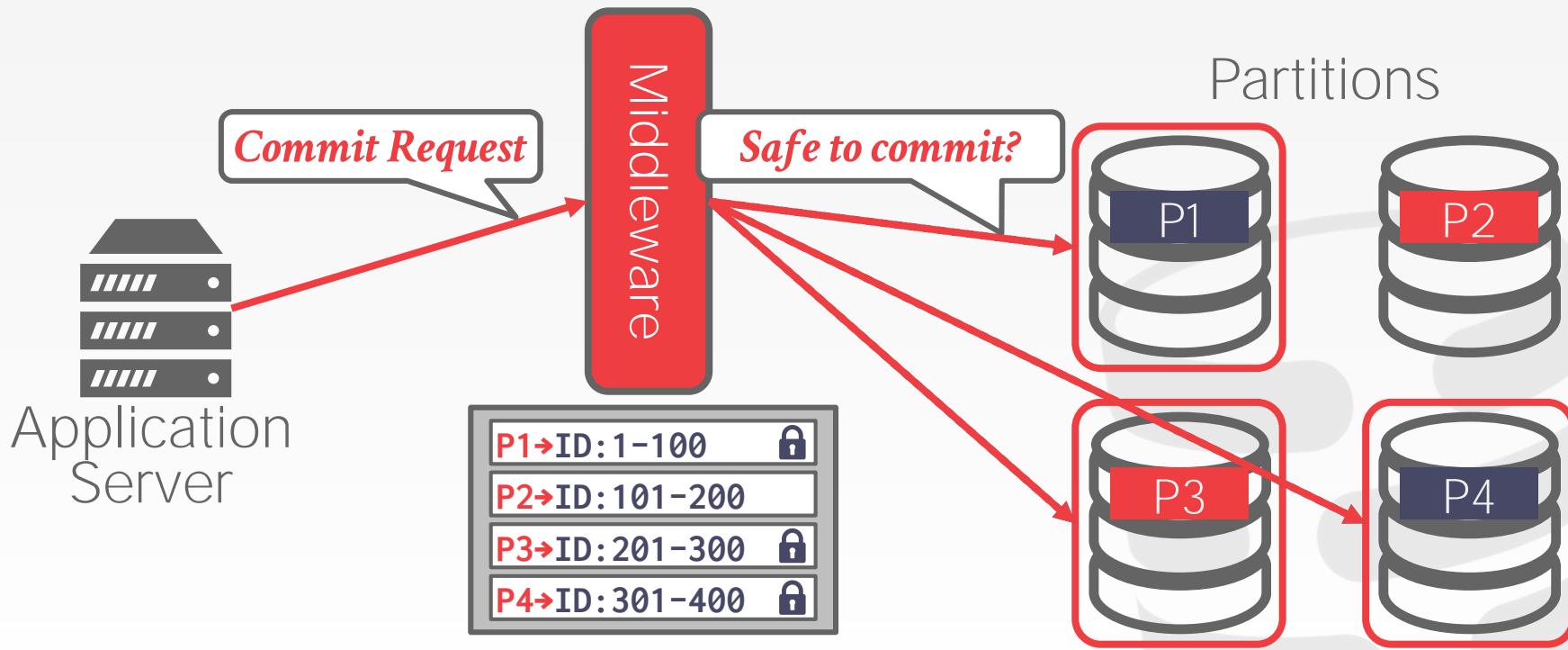
Originally developed in the 1970-80s to provide txns between terminals and mainframe databases.
→ Examples: ATMs, Airline Reservations.

Many DBMSs now support the same functionality internally.

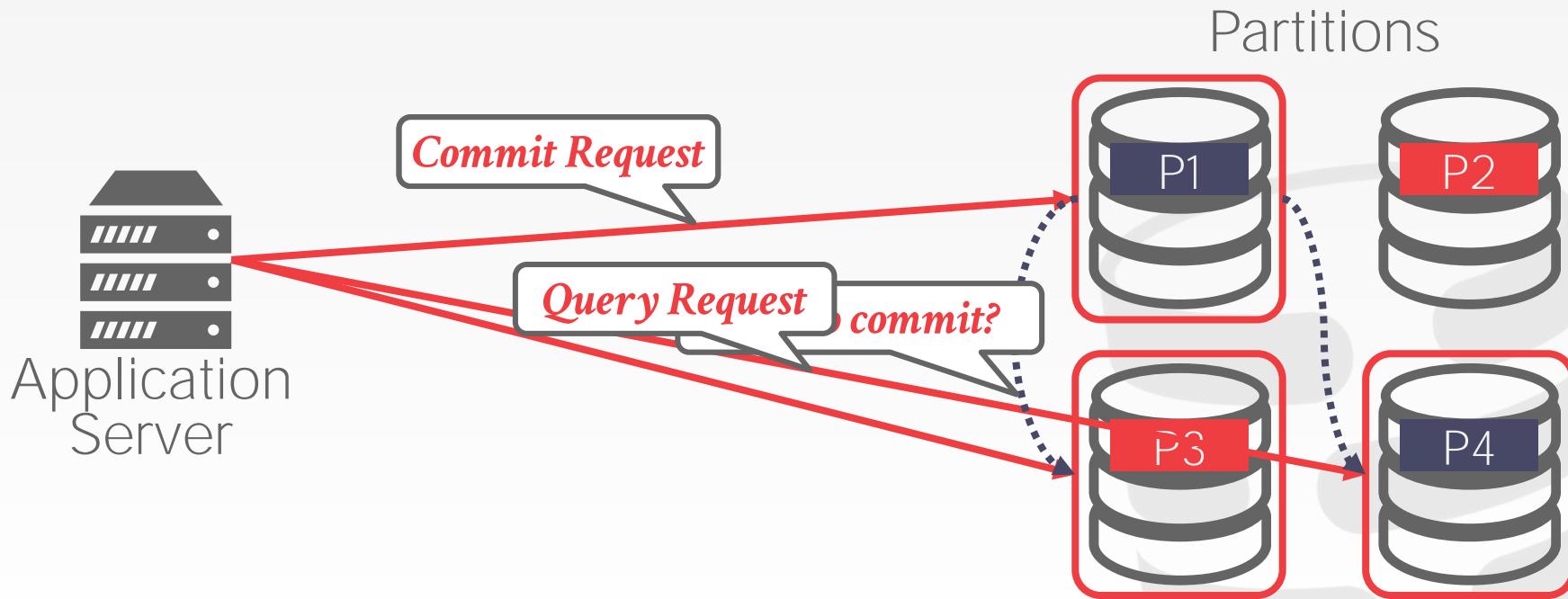
CENTRALIZED COORDINATOR



CENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DISTRIBUTED CONCURRENCY CONTROL

Need to allow multiple txns to execute simultaneously across multiple nodes.

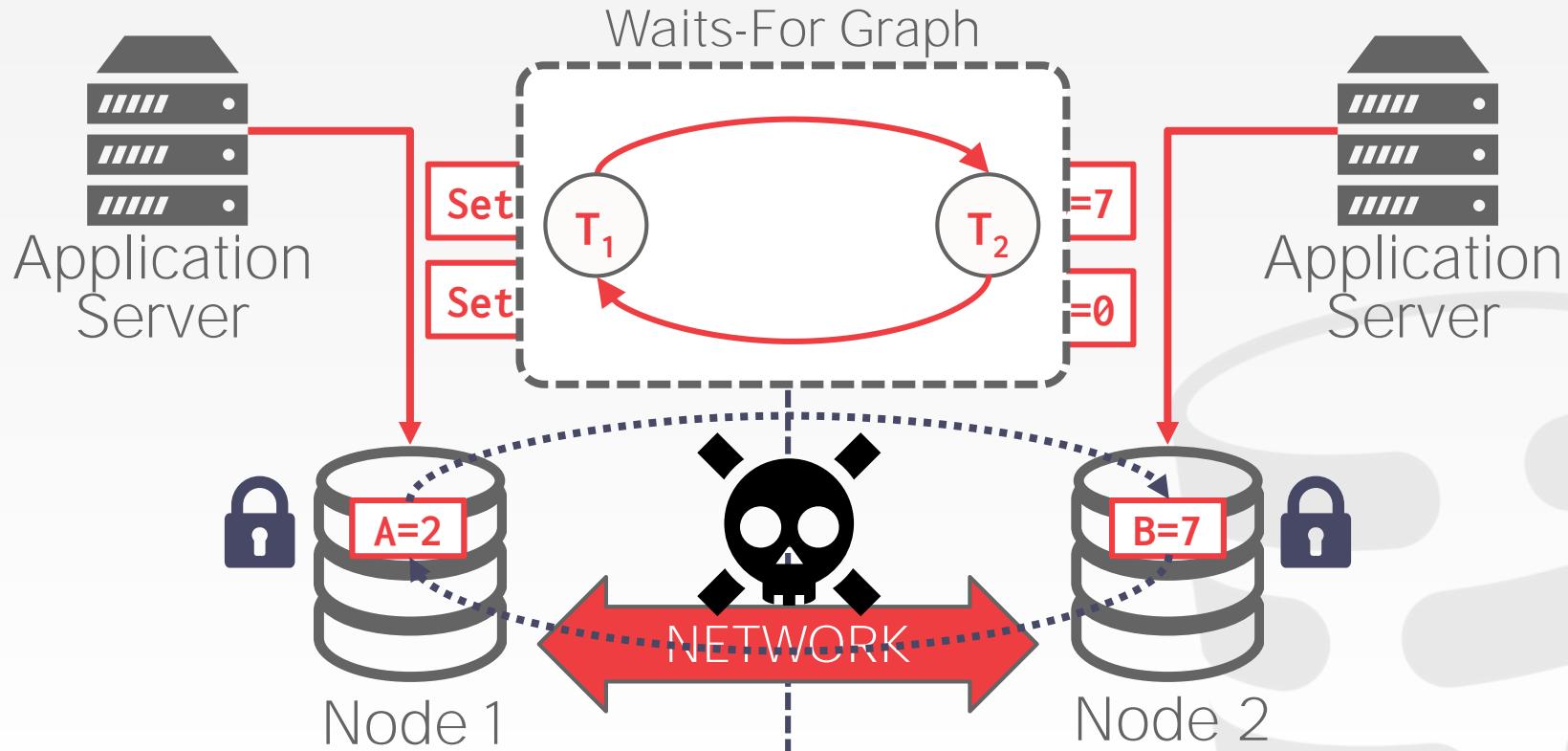
→ Many of the same protocols from single-node DBMSs can be adapted.

This is harder because of:

- Replication.
- Network Communication Overhead.
- Node Failures.
- Clock Skew.



DISTRIBUTED 2PL



CONCLUSION

I have barely scratched the surface on distributed database systems...

It is hard to get right.

More info (and humiliation):
→ [Kyle Kingsbury's Jepsen Project](#)



NEXT CLASS

Distributed OLTP Systems

Replication

CAP Theorem

Real-World Examples



23

Distributed OLTP Databases



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

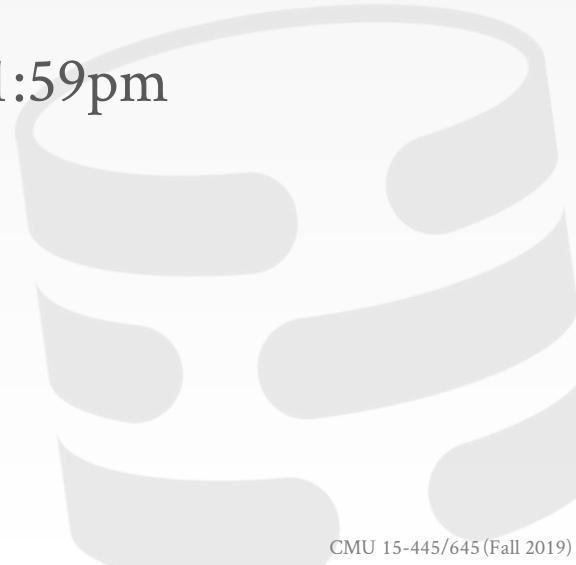
ADMINISTRIVIA

Homework #5: Monday Dec 3rd @ 11:59pm

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 10th @ 11:59pm

Final Exam: Monday Dec 9th @ 5:30pm



LAST CLASS

System Architectures

→ Shared-Memory, Shared-Disk, Shared-Nothing

Partitioning/Sharding

→ Hash, Range, Round Robin

Transaction Coordination

→ Centralized vs. Decentralized



OLTP VS. OLAP

On-line Transaction Processing (OLTP):

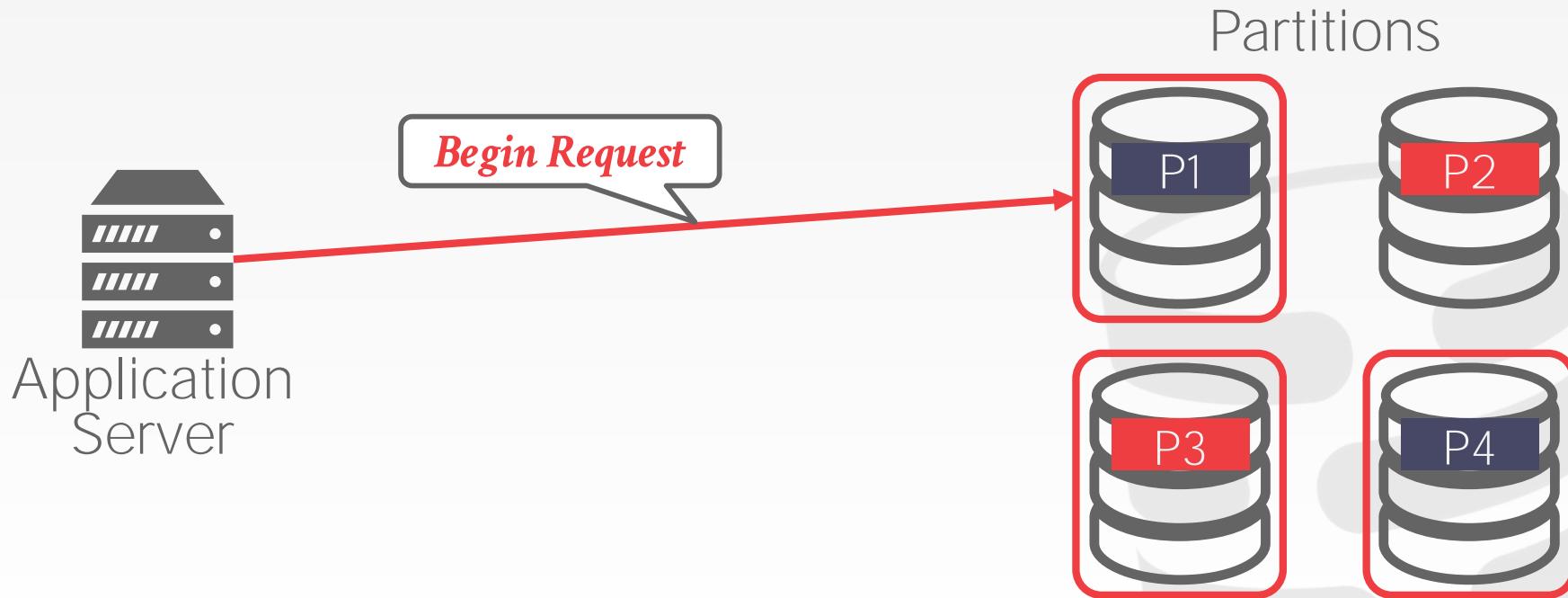
- Short-lived read/write txns.
- Small footprint.
- Repetitive operations.

On-line Analytical Processing (OLAP):

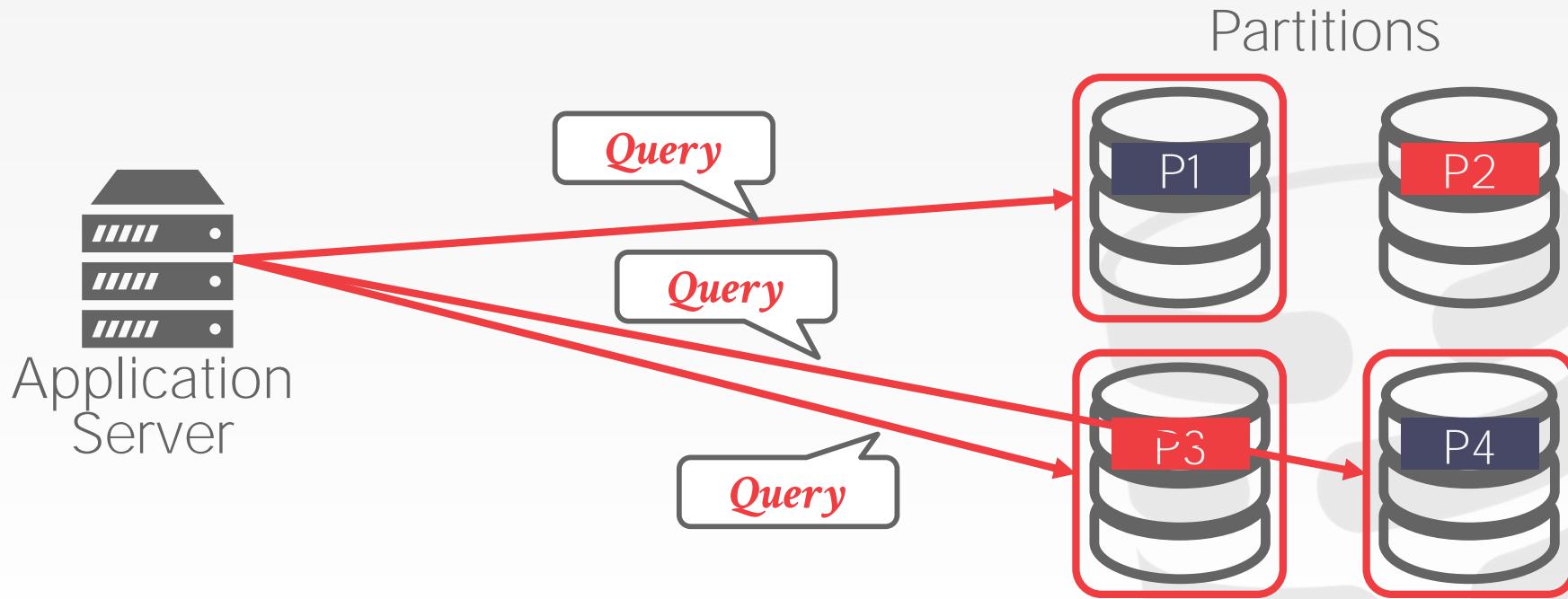
- Long-running, read-only queries.
- Complex joins.
- Exploratory queries.



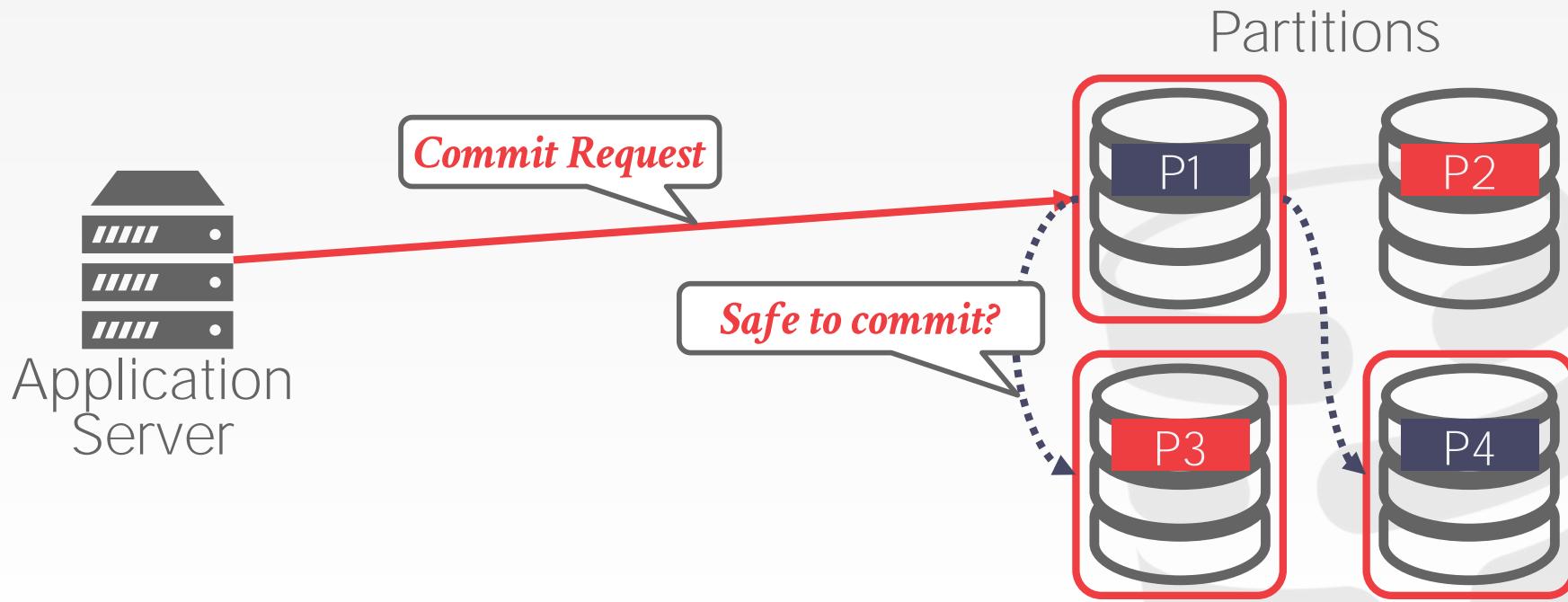
DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



OBSERVATION

We have not discussed how to ensure that all nodes agree to commit a txn and then to make sure it does commit if we decide that it should.

- What happens if a node fails?
- What happens if our messages show up late?
- What happens if we don't wait for every node to agree?



IMPORTANT ASSUMPTION

We can assume that all nodes in a distributed DBMS are well-behaved and under the same administrative domain.

→ If we tell a node to commit a txn, then it will commit the txn (if there is not a failure).

If you do not trust the other nodes in a distributed DBMS, then you need to use a Byzantine Fault Tolerant protocol for txns (blockchain).

TODAY'S AGENDA

Atomic Commit Protocols

Replication

Consistency Issues (CAP)

Federated Databases



ATOMIC COMMIT PROTOCOL

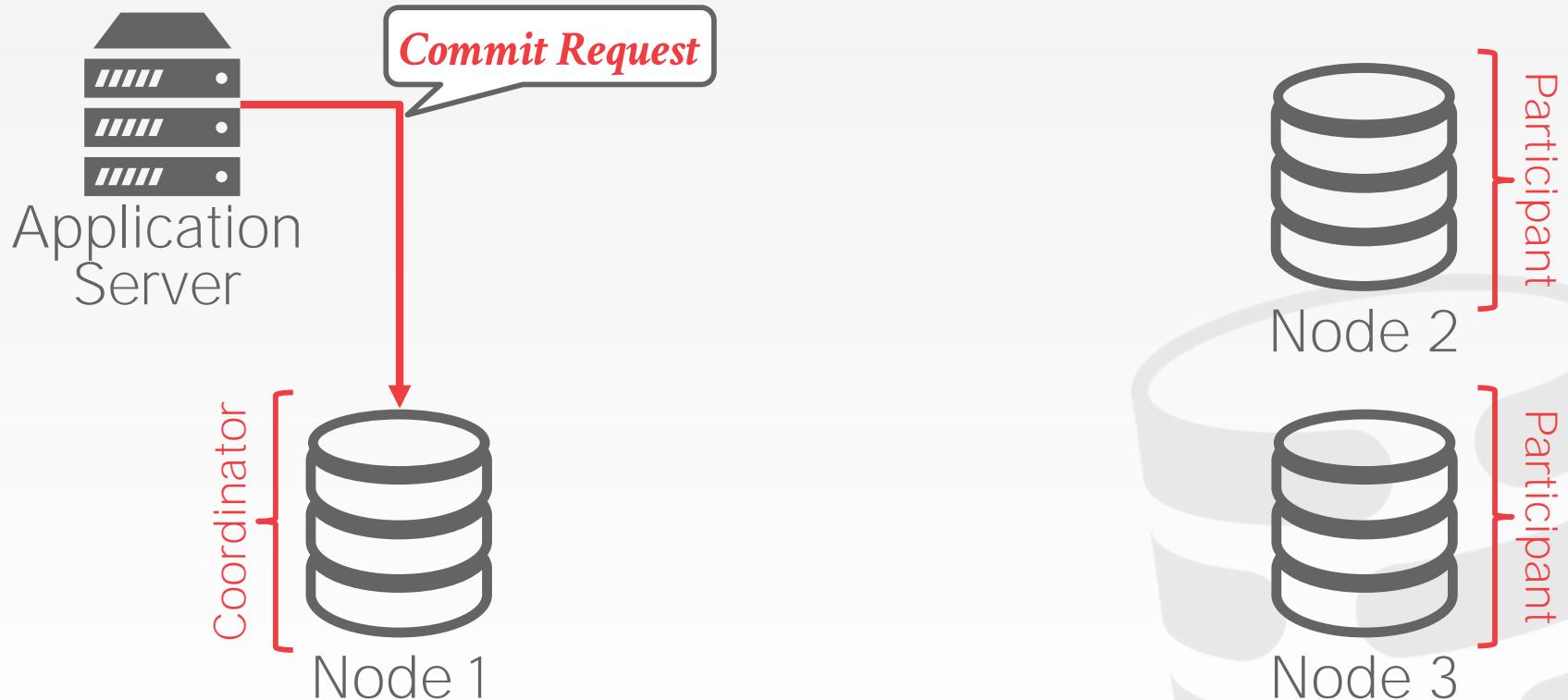
When a multi-node txn finishes, the DBMS needs to ask all the nodes involved whether it is safe to commit.

Examples:

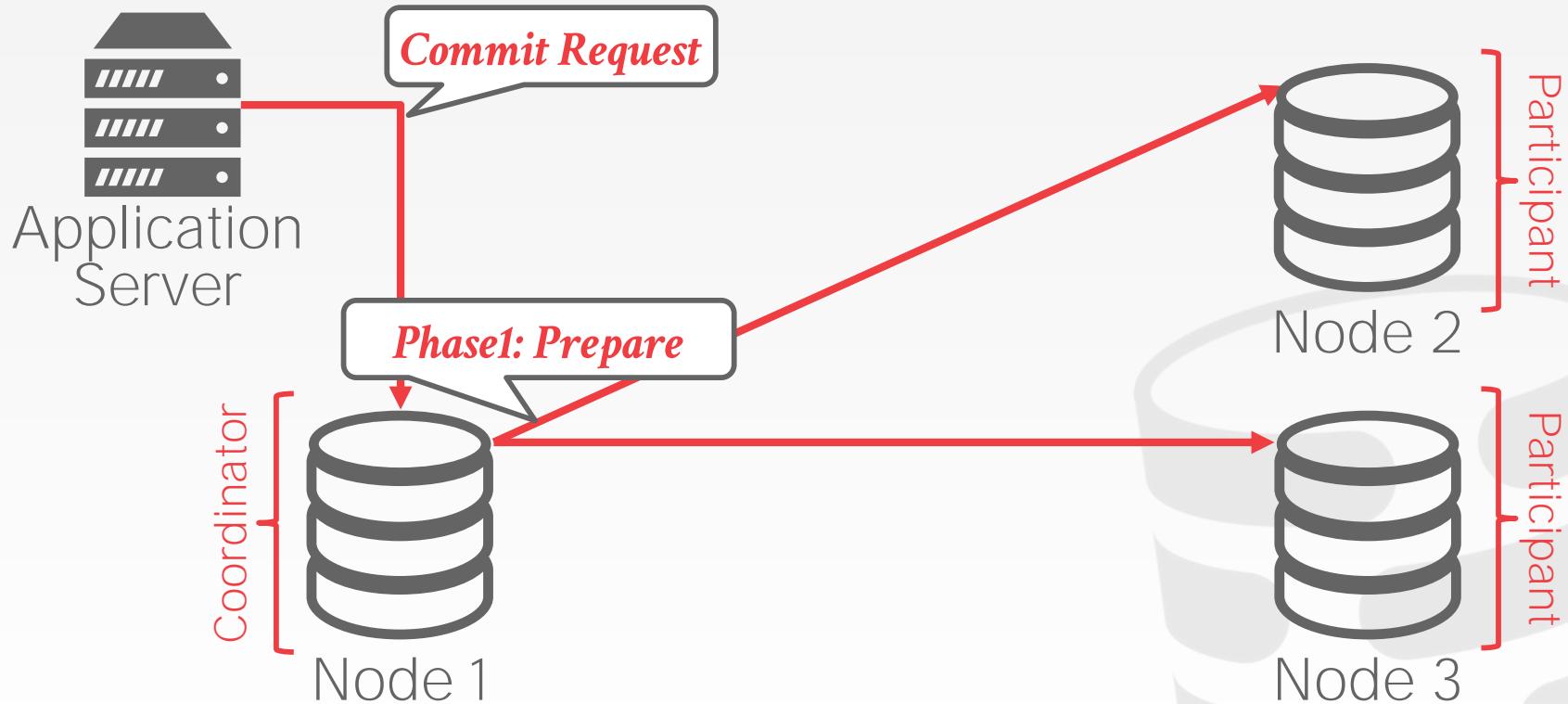
- Two-Phase Commit
- Three-Phase Commit (not used)
- Paxos
- Raft
- ZAB (Apache Zookeeper)
- Viewstamped Replication



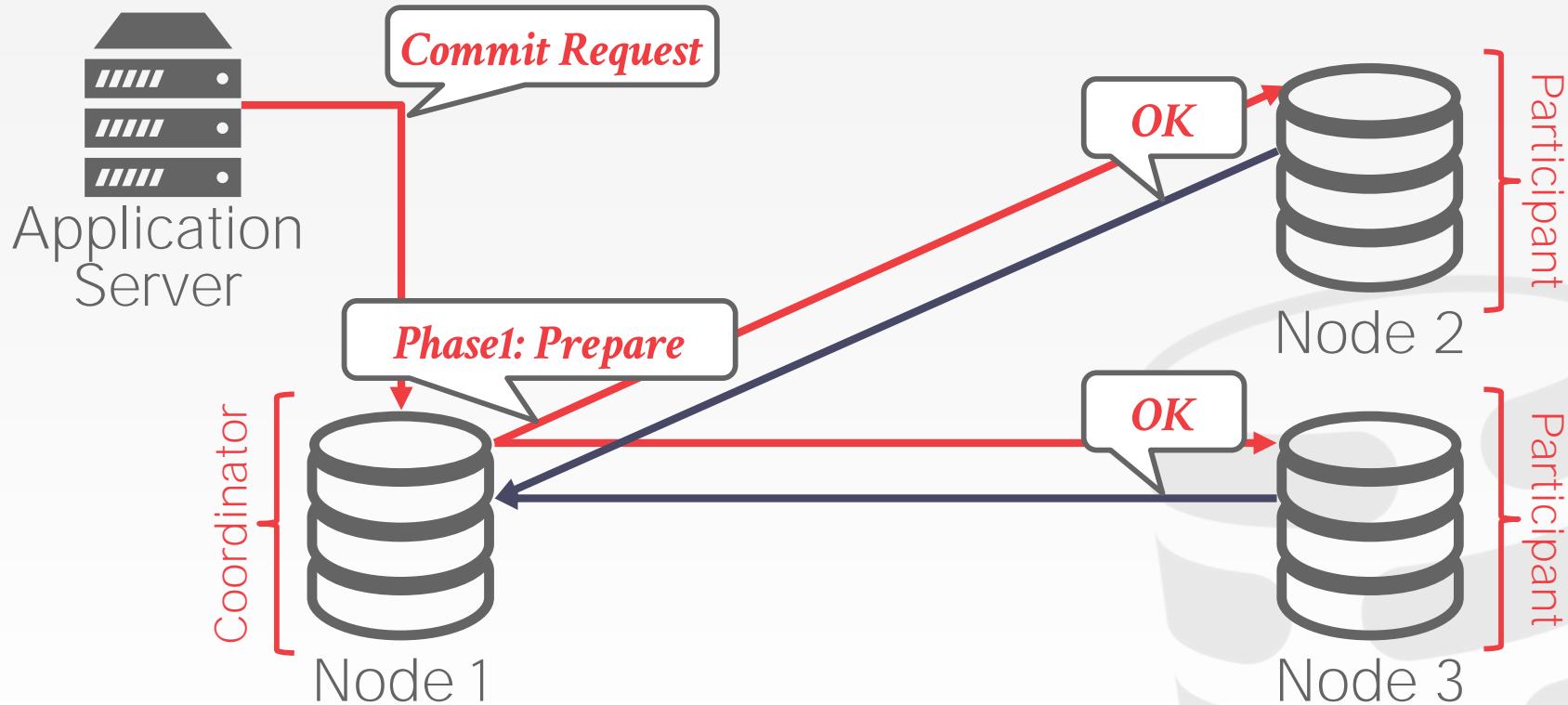
TWO-PHASE COMMIT (SUCCESS)



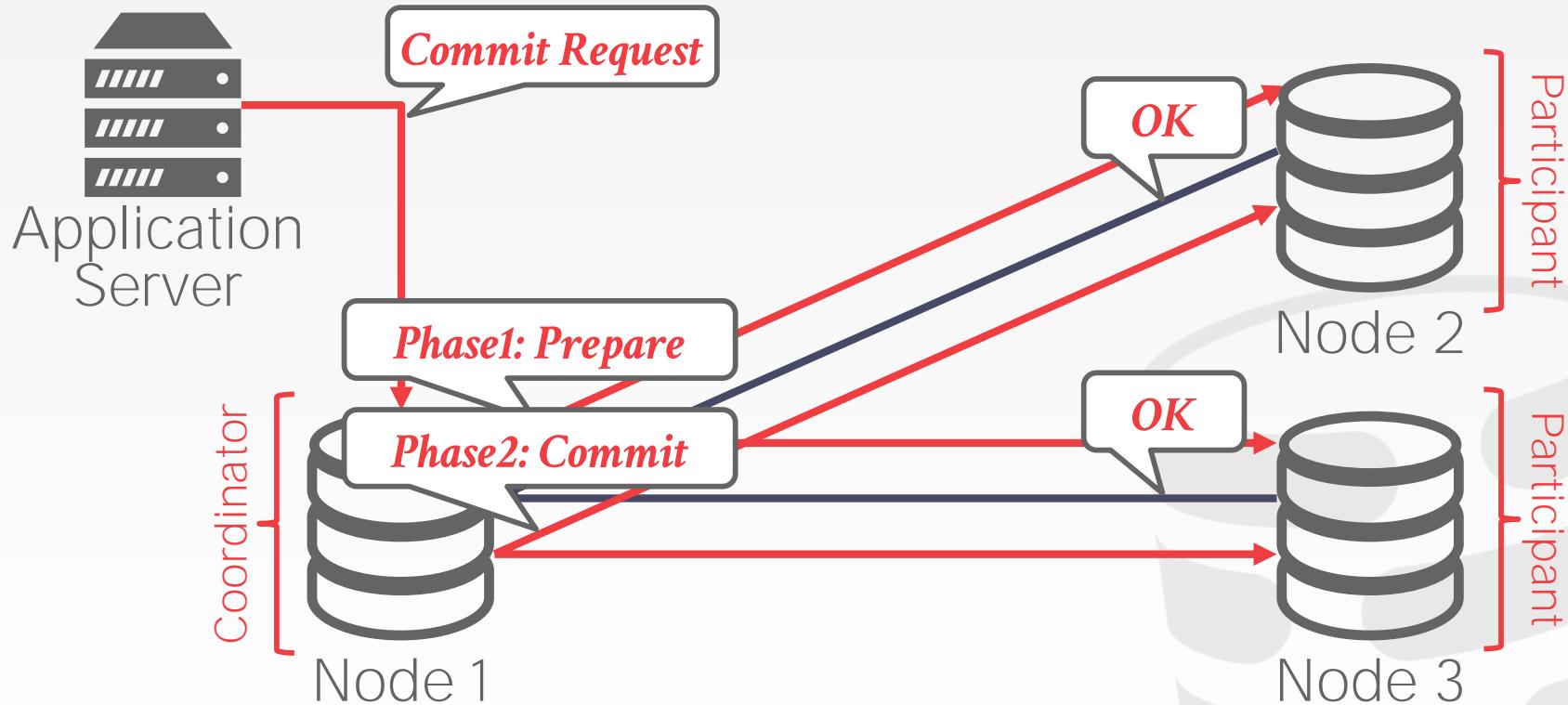
TWO-PHASE COMMIT (SUCCESS)



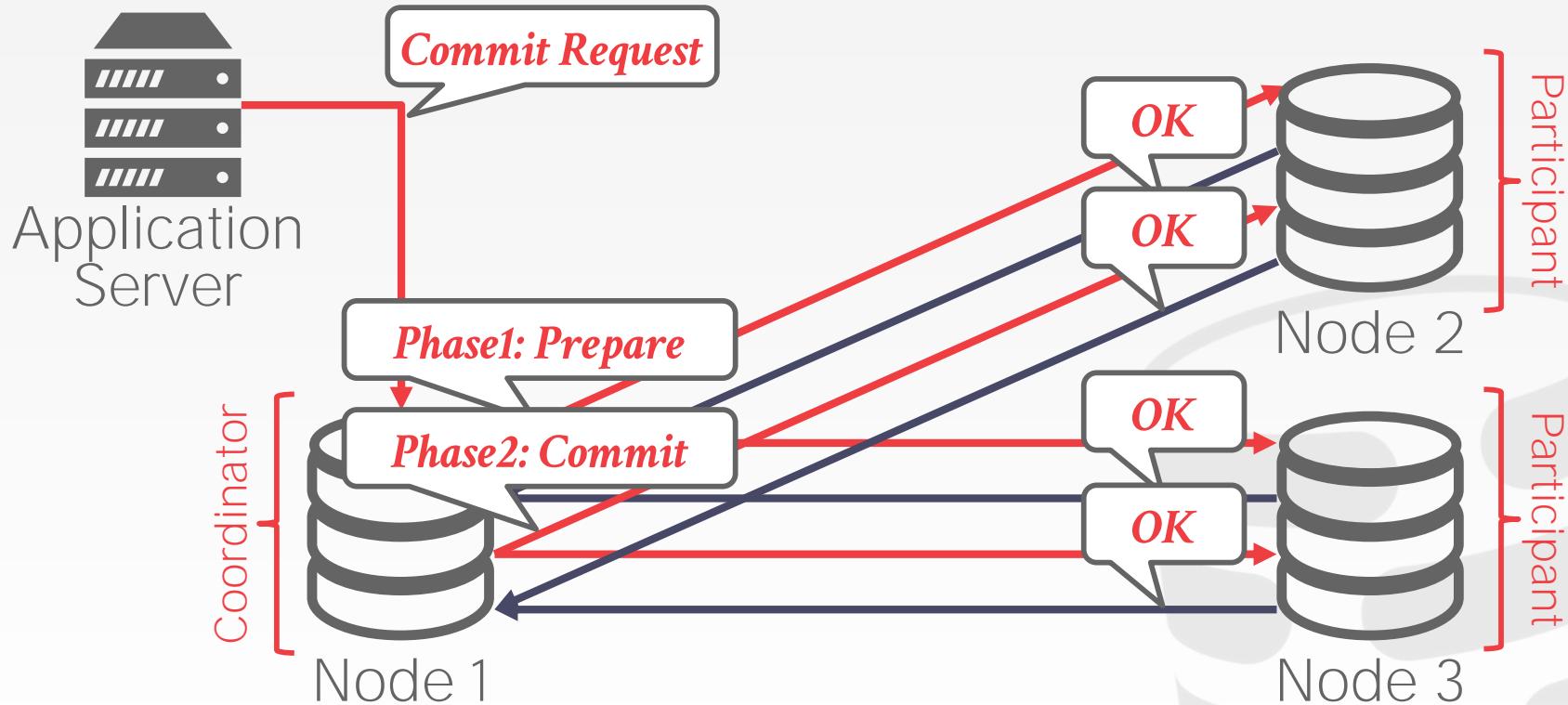
TWO-PHASE COMMIT (SUCCESS)



TWO-PHASE COMMIT (SUCCESS)



TWO-PHASE COMMIT (SUCCESS)



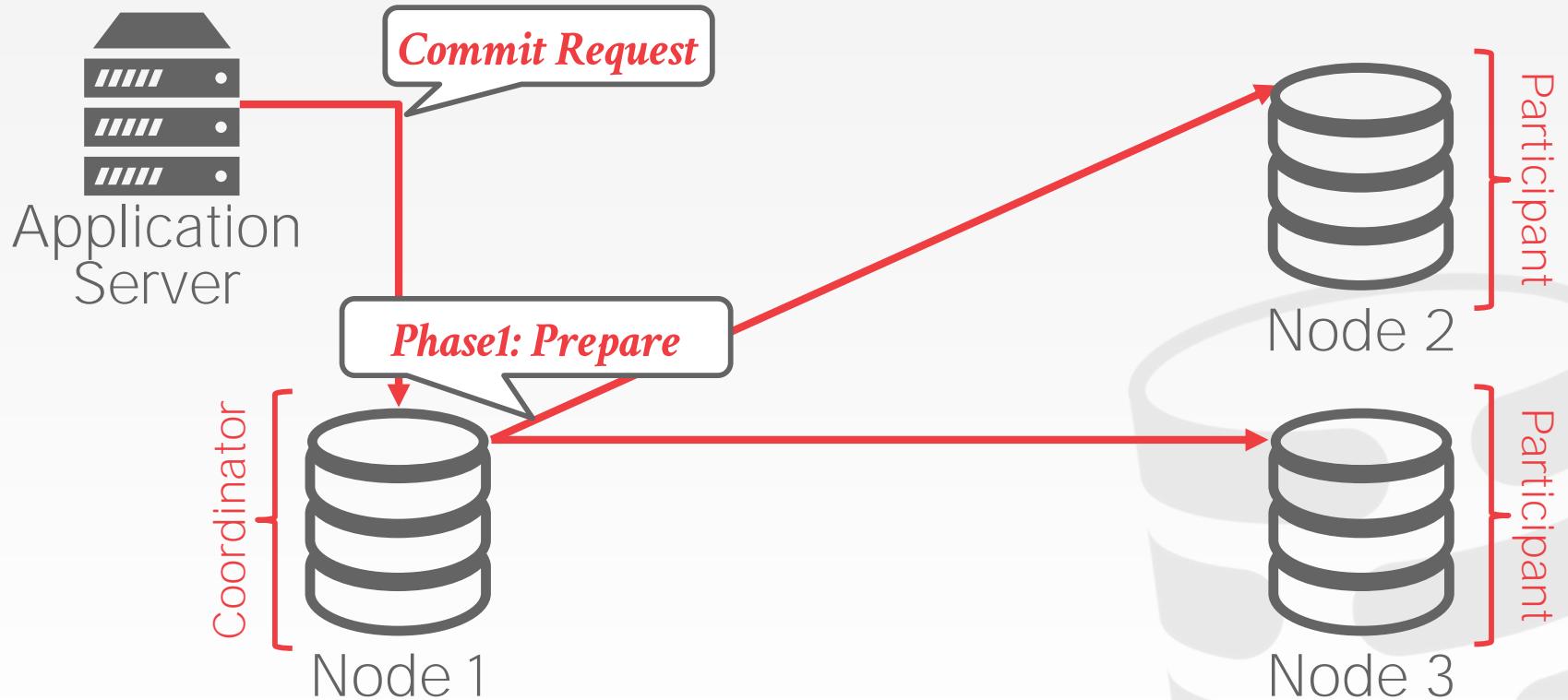
TWO-PHASE COMMIT (SUCCESS)



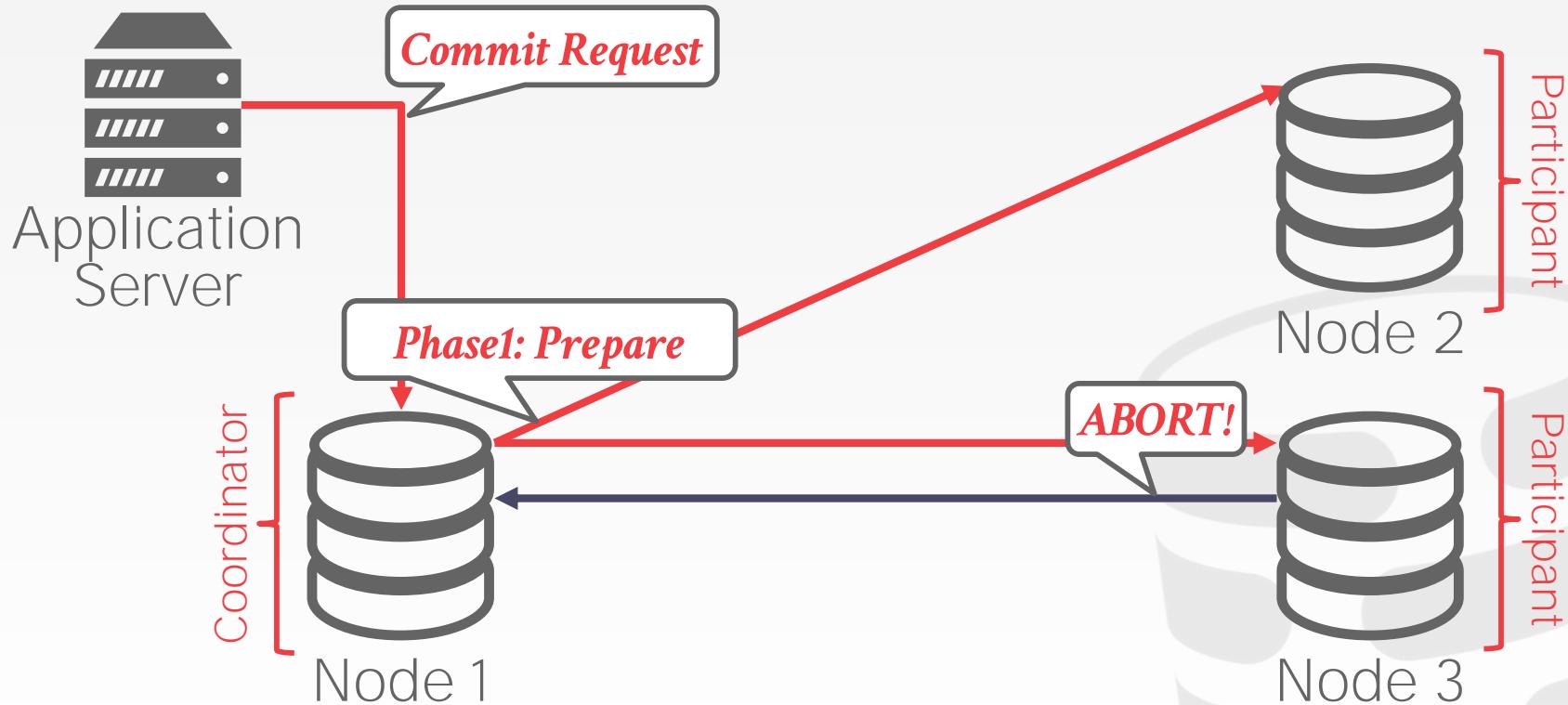
TWO-PHASE COMMIT (ABORT)



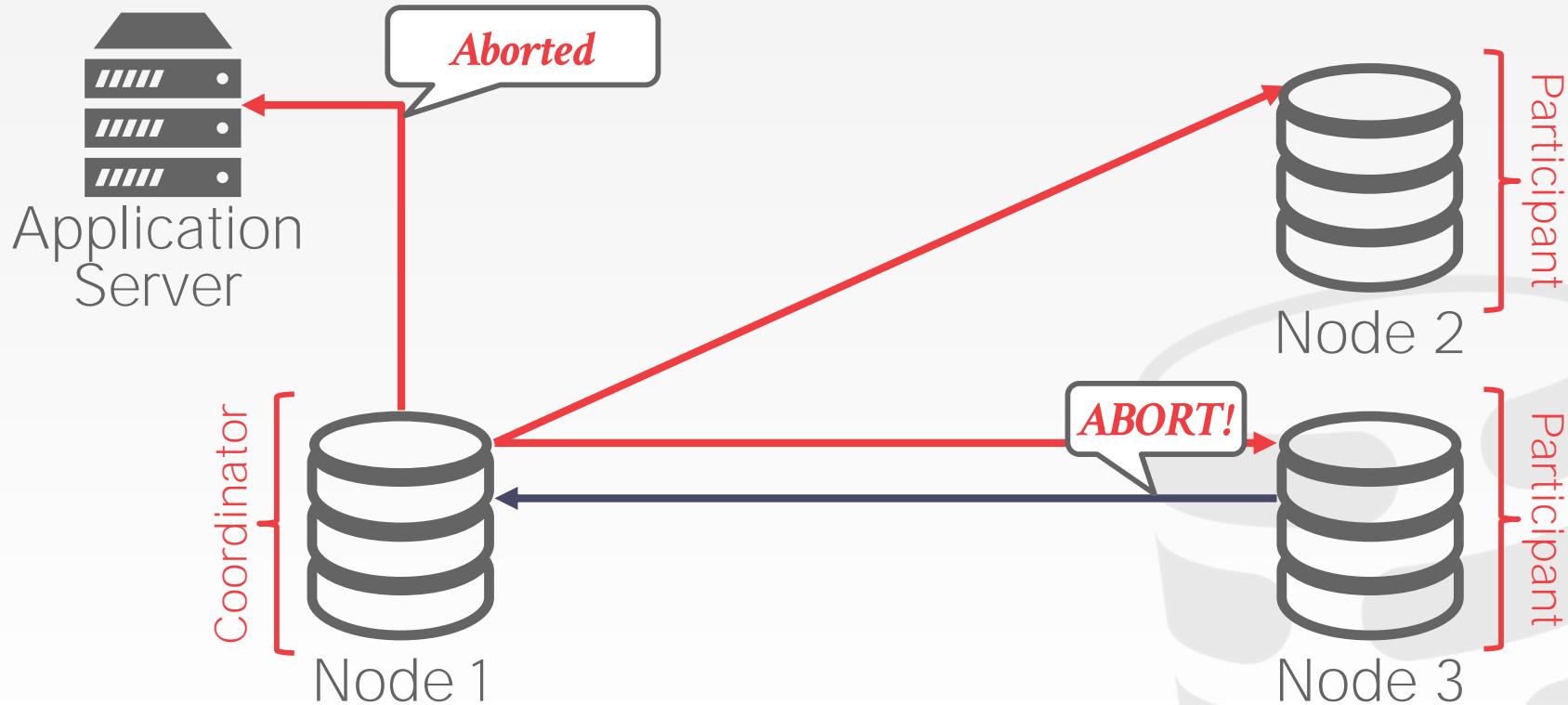
TWO-PHASE COMMIT (ABORT)



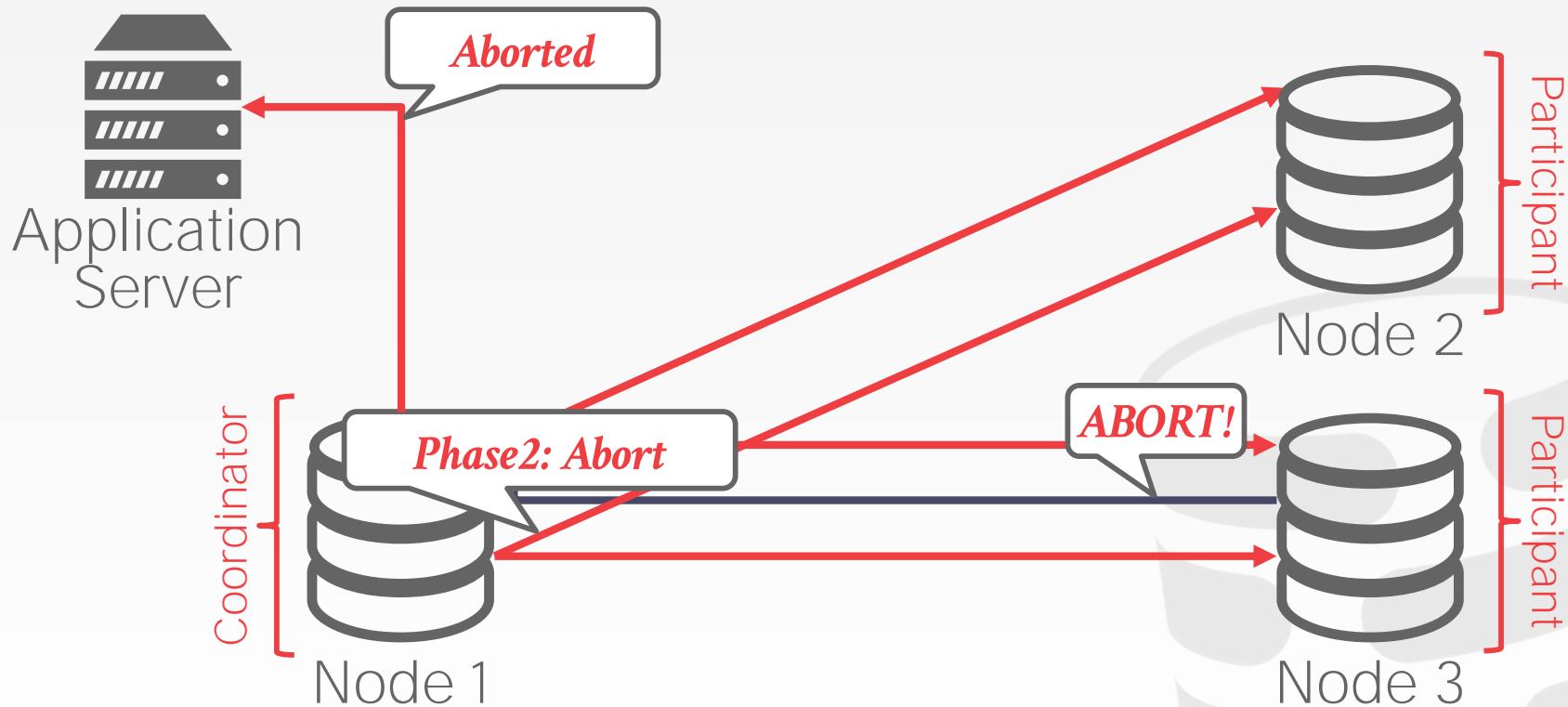
TWO-PHASE COMMIT (ABORT)



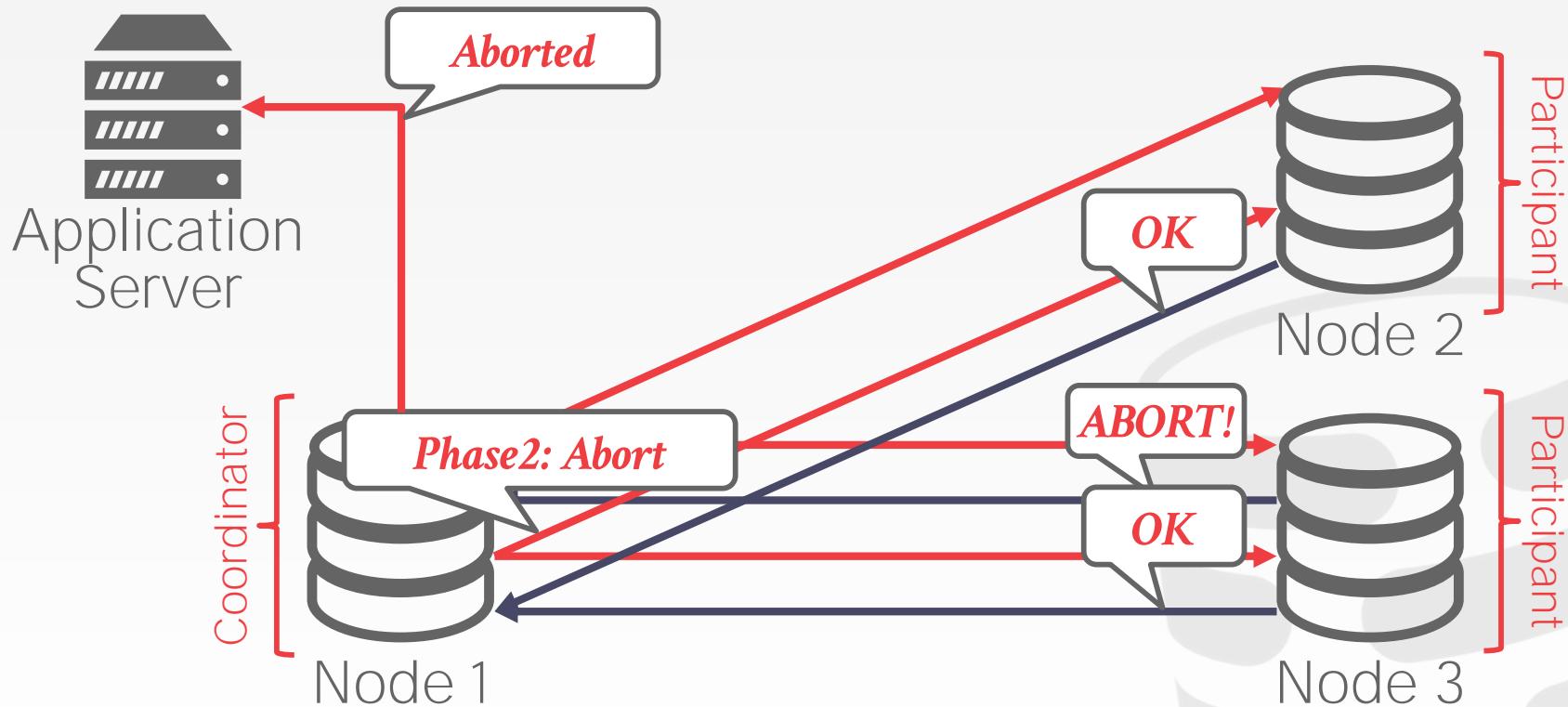
TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



2PC OPTIMIZATIONS

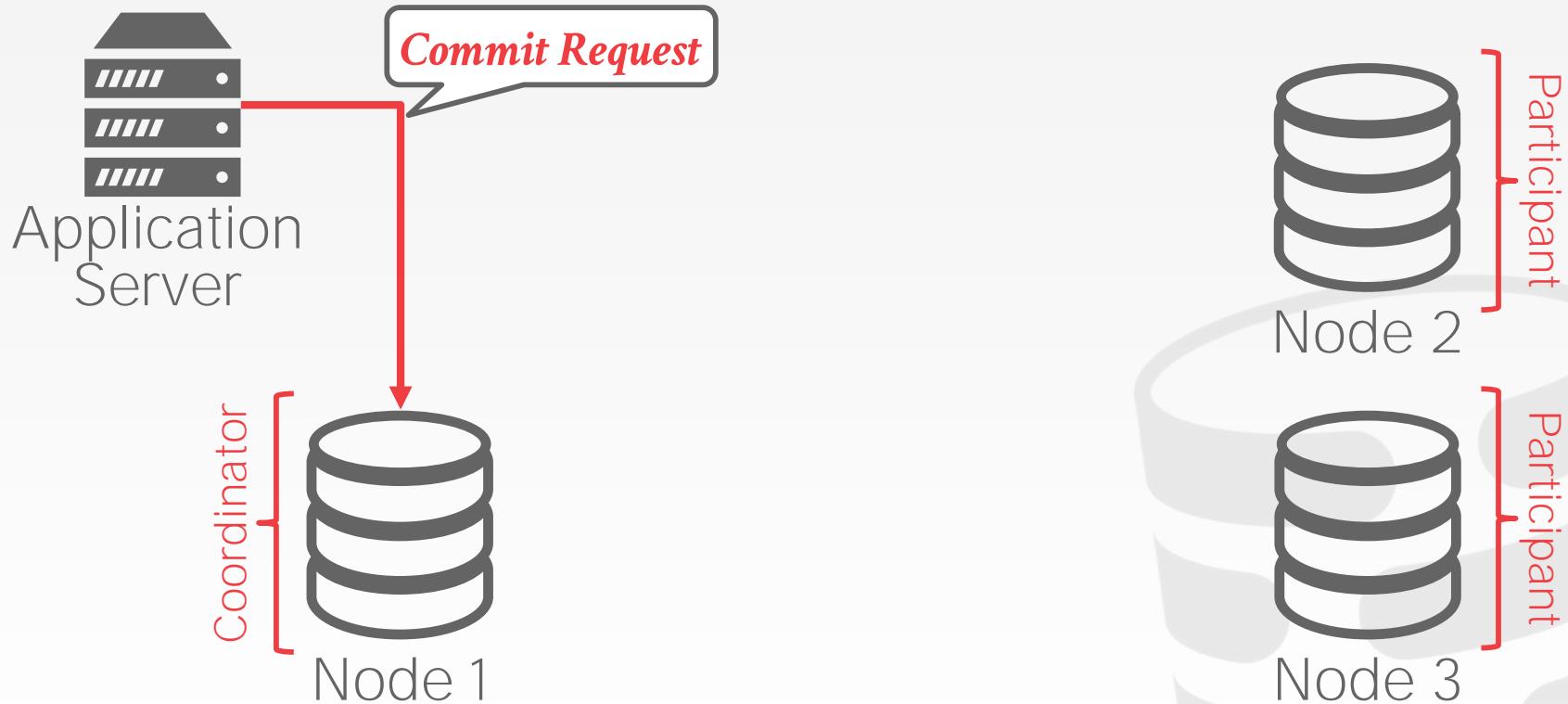
Early Prepare Voting

- If you send a query to a remote node that you know will be the last one you execute there, then that node will also return their vote for the prepare phase with the query result.

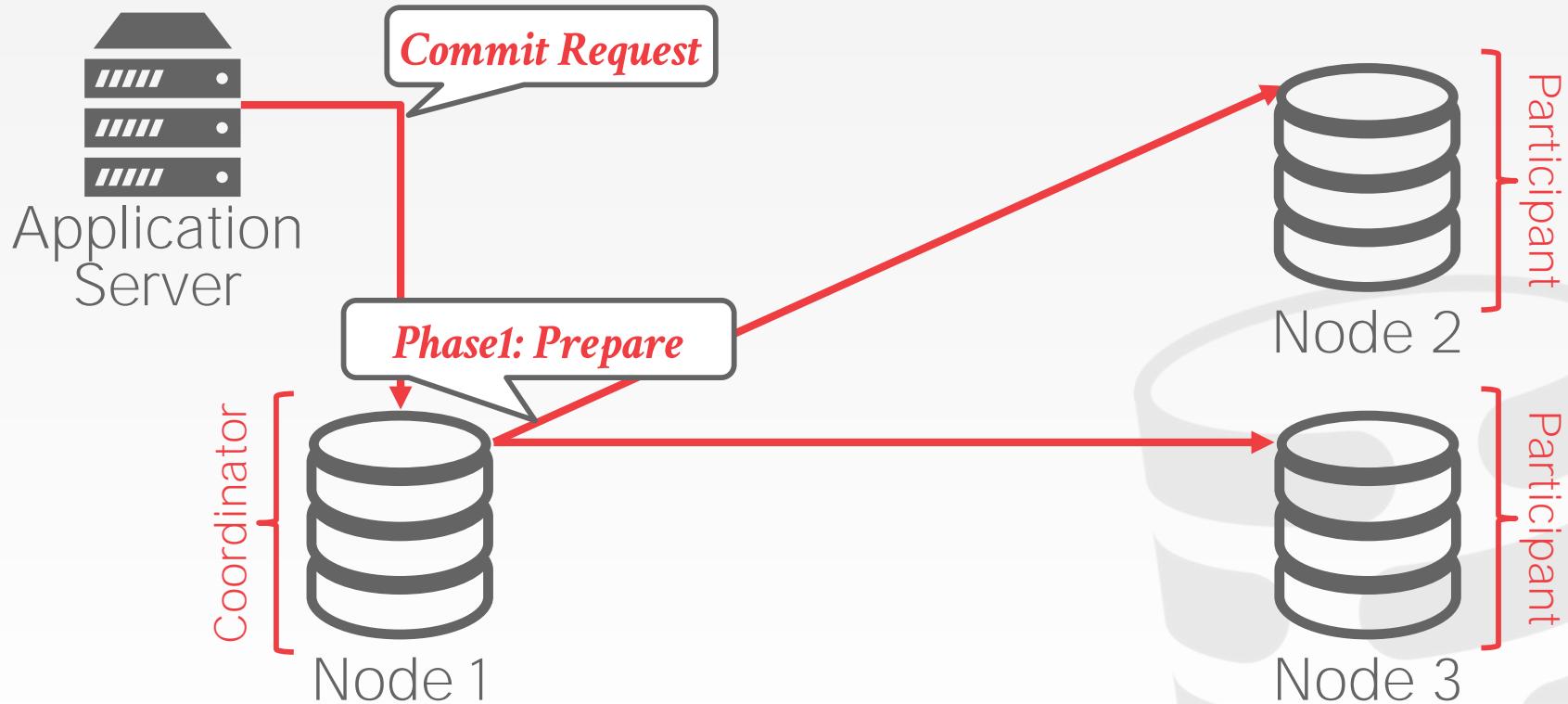
Early Acknowledgement After Prepare

- If all nodes vote to commit a txn, the coordinator can send the client an acknowledgement that their txn was successful before the commit phase finishes.

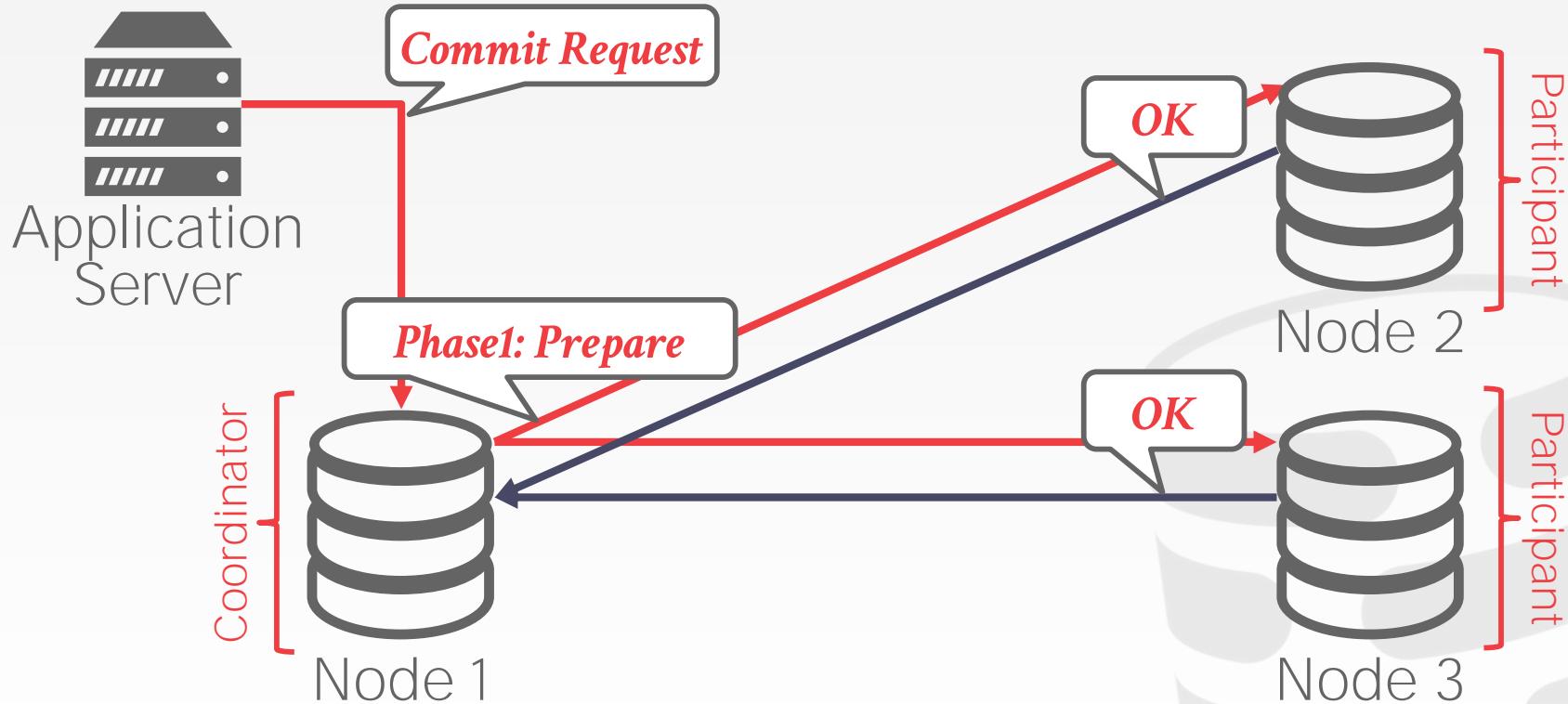
EARLY ACKNOWLEDGEMENT



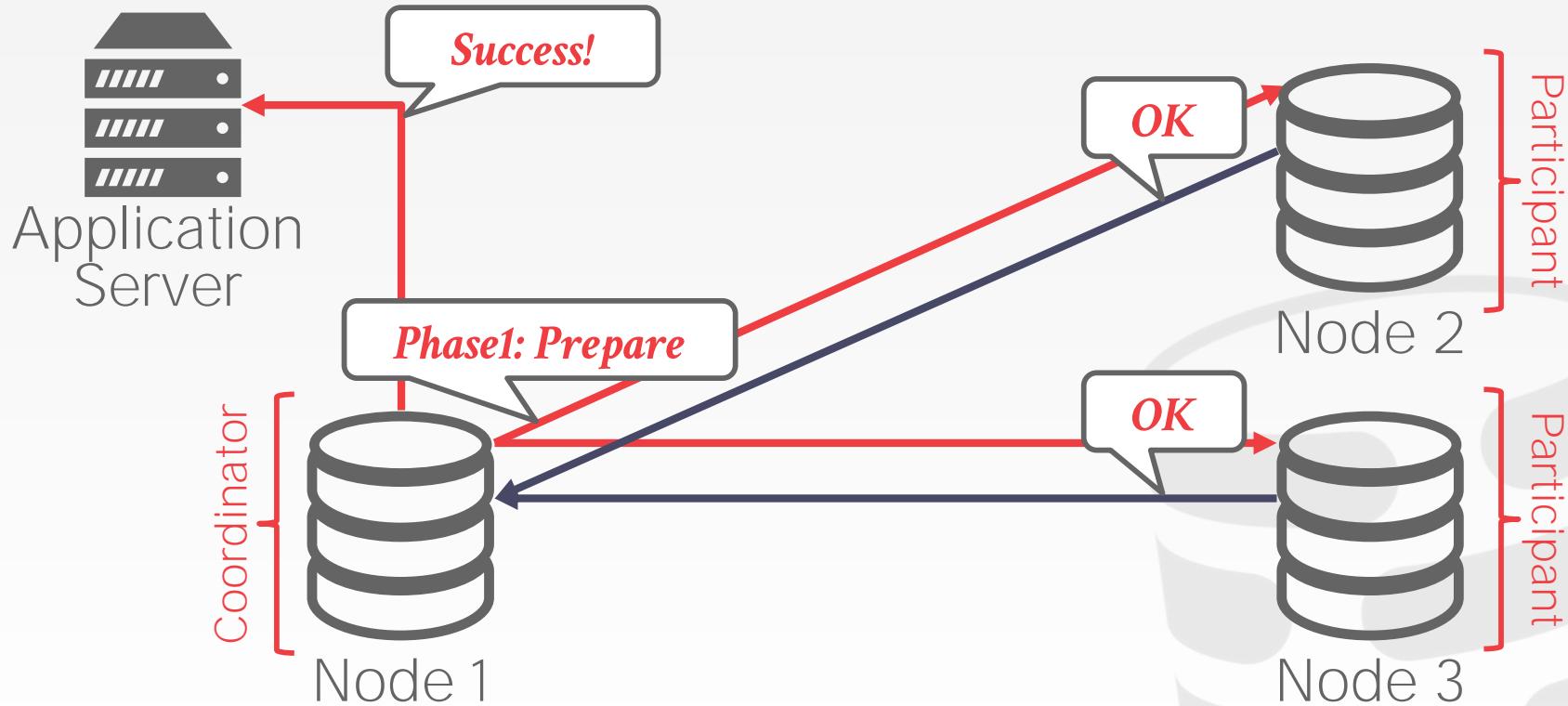
EARLY ACKNOWLEDGEMENT



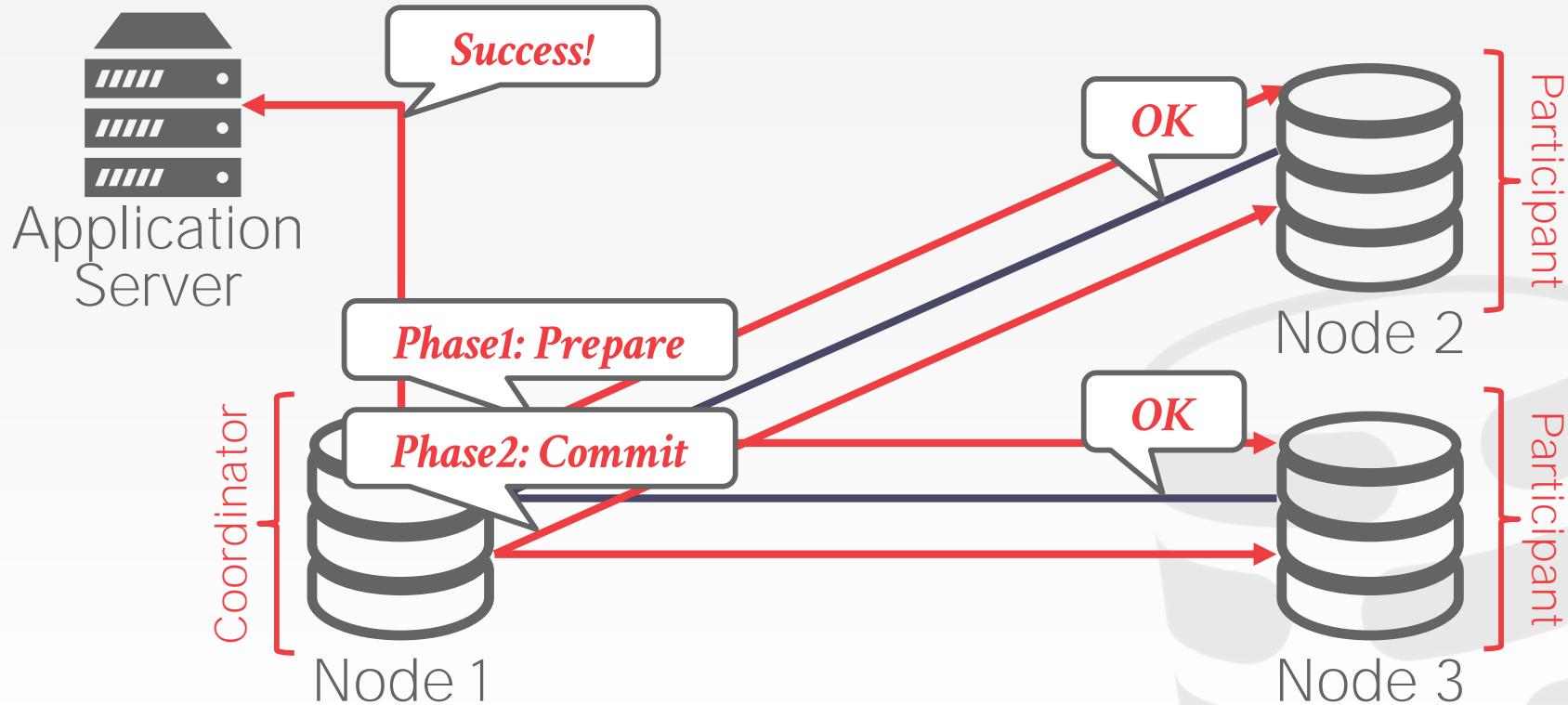
EARLY ACKNOWLEDGEMENT



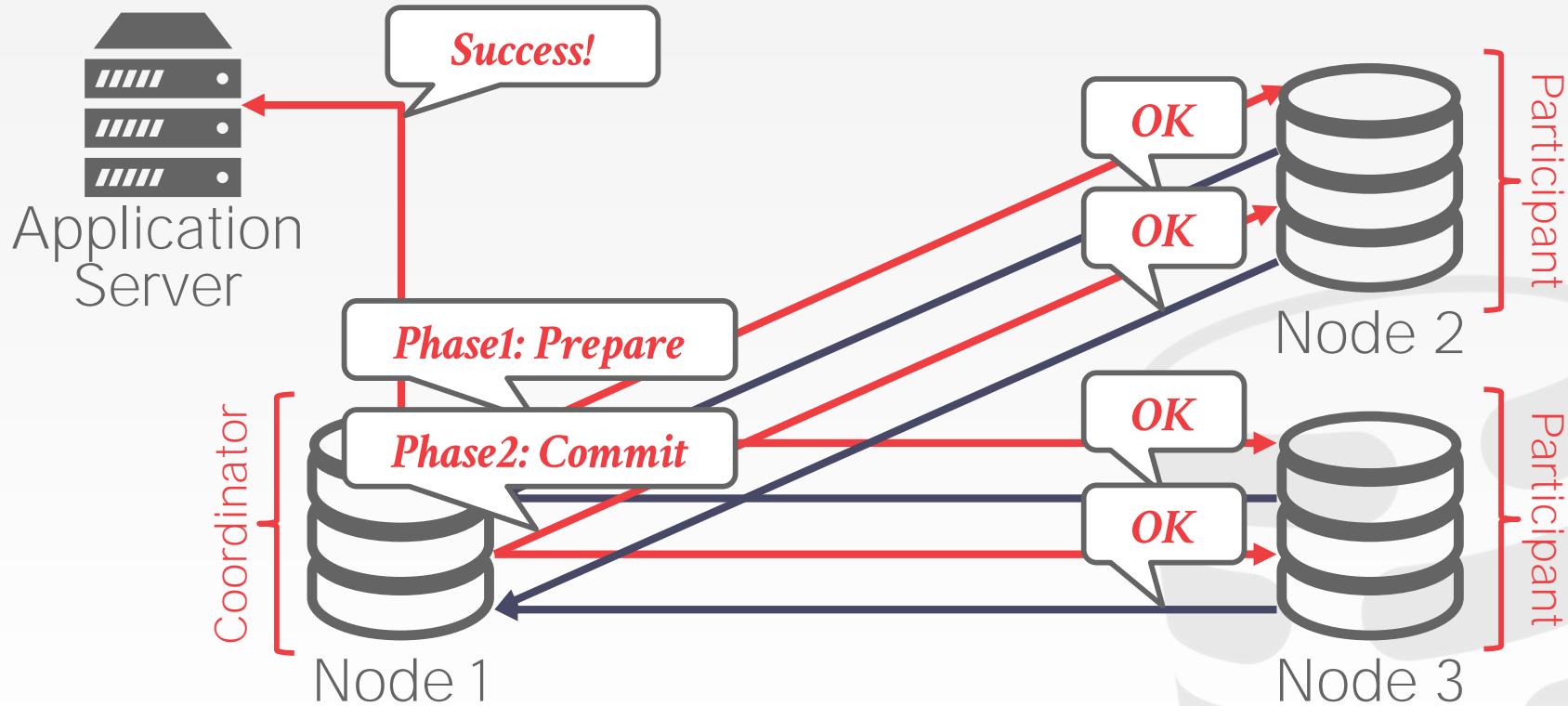
EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



TWO-PHASE COMMIT

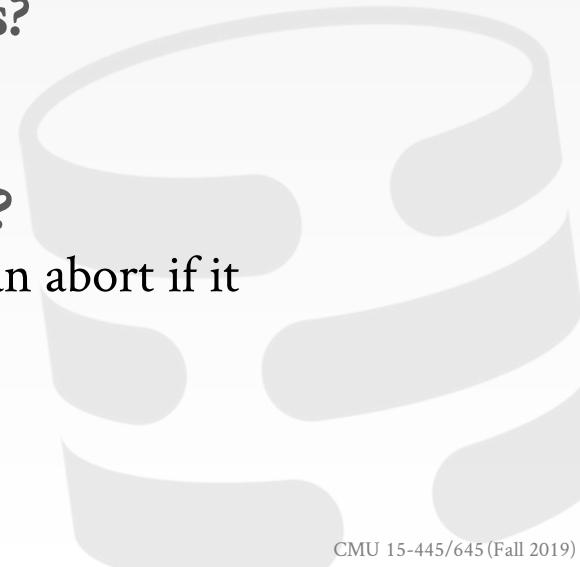
Each node records the outcome of each phase in a non-volatile storage log.

What happens if coordinator crashes?

→ Participants must decide what to do.

What happens if participant crashes?

→ Coordinator assumes that it responded with an abort if it hasn't sent an acknowledgement yet.



PAXOS

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays in the best case.

The Part-Time Parliament

LESLIE LAMPURT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record in their portmanteaus for the chamber and the forgotten recesses of memory. The Paxos parliament's procedure provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [Computer-Communications Networks]; Distributed Systems; Network operating systems; D4.5 [Operating Systems]: Reliability—Fault-tolerance; F.1 [Administrative Data Processing]; Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author claims to be an archeologist with only a passing interest in computer science. This is unfortunate, even though the obscure ancient Paxos civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. In fact, the improvements the Paxons made to their protocol appear to be unknown in the systems literature.

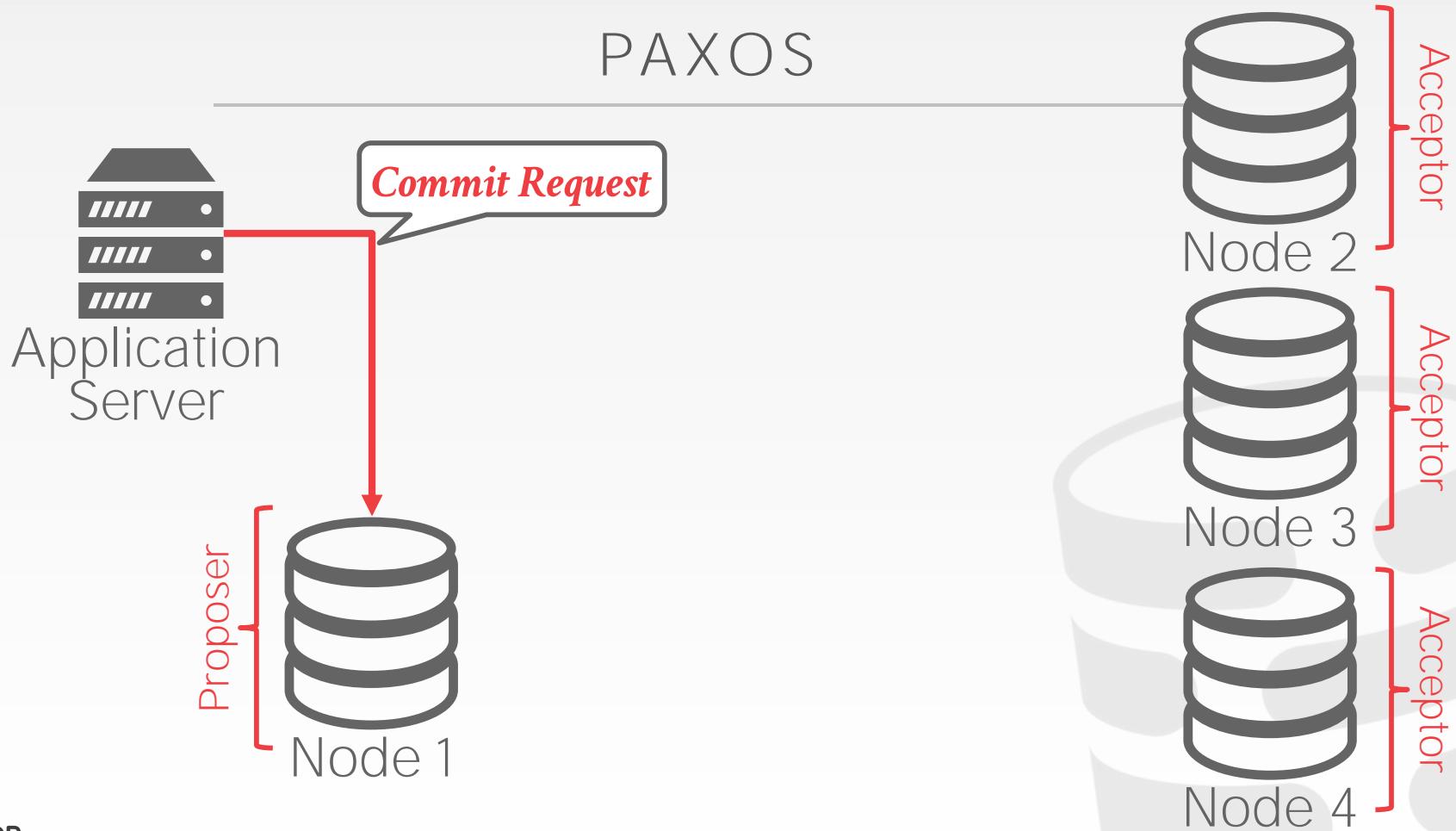
The author does give a brief discussion of the Paxos Parliament's relevance to distributed systems. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lamport [1999]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

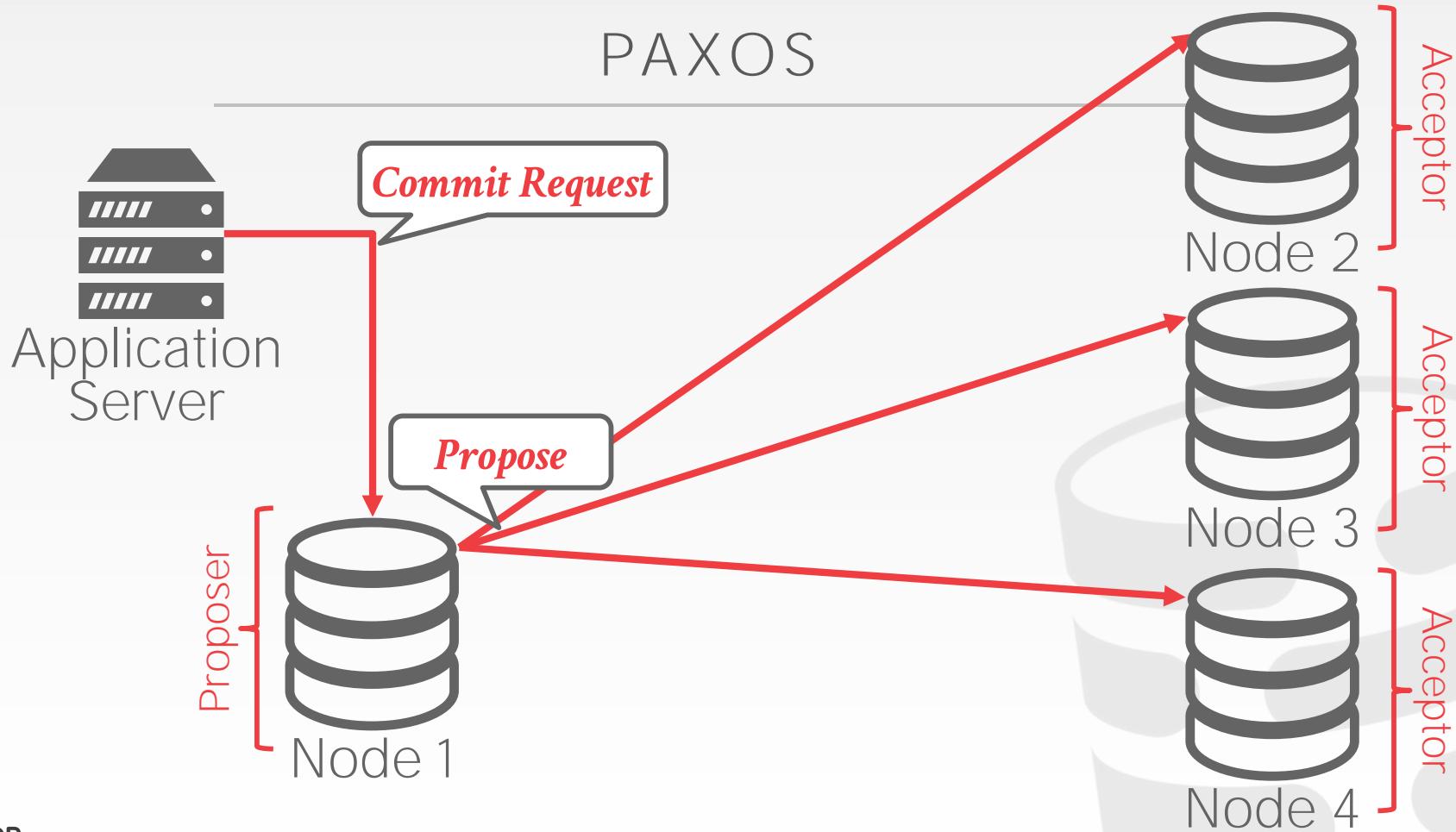
Keith Marzullo
University of California, San Diego

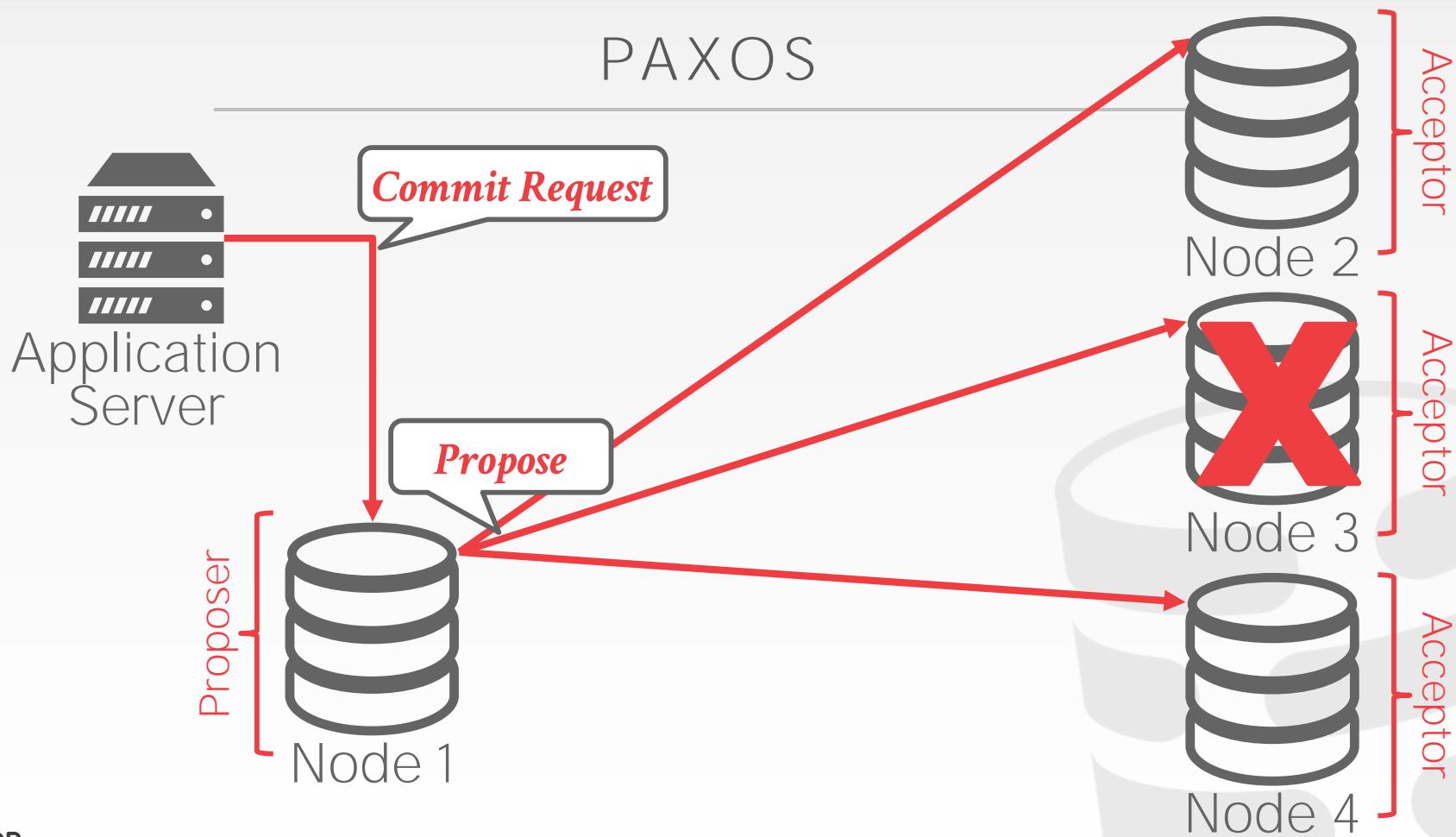
Authors' address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.

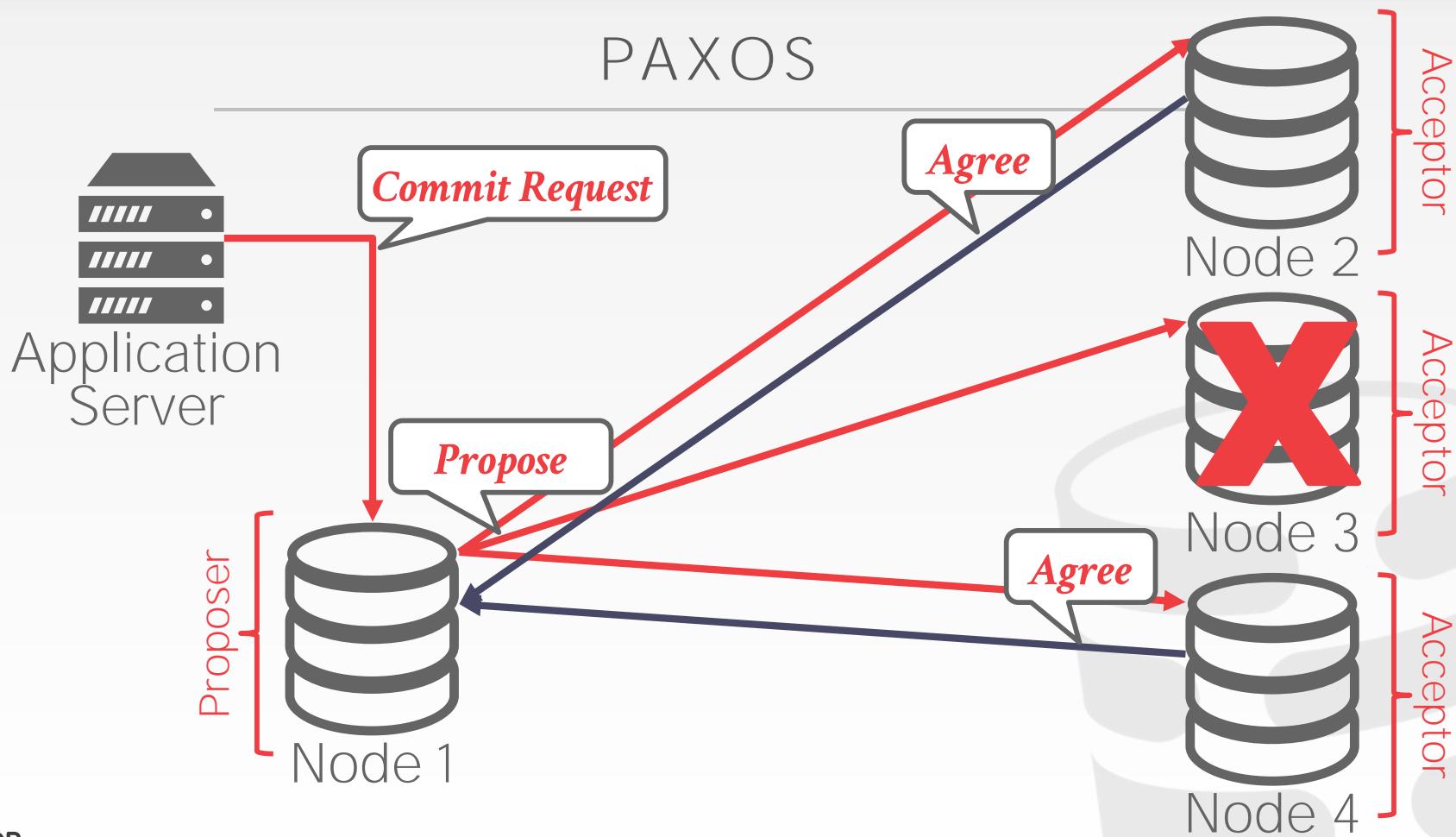
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

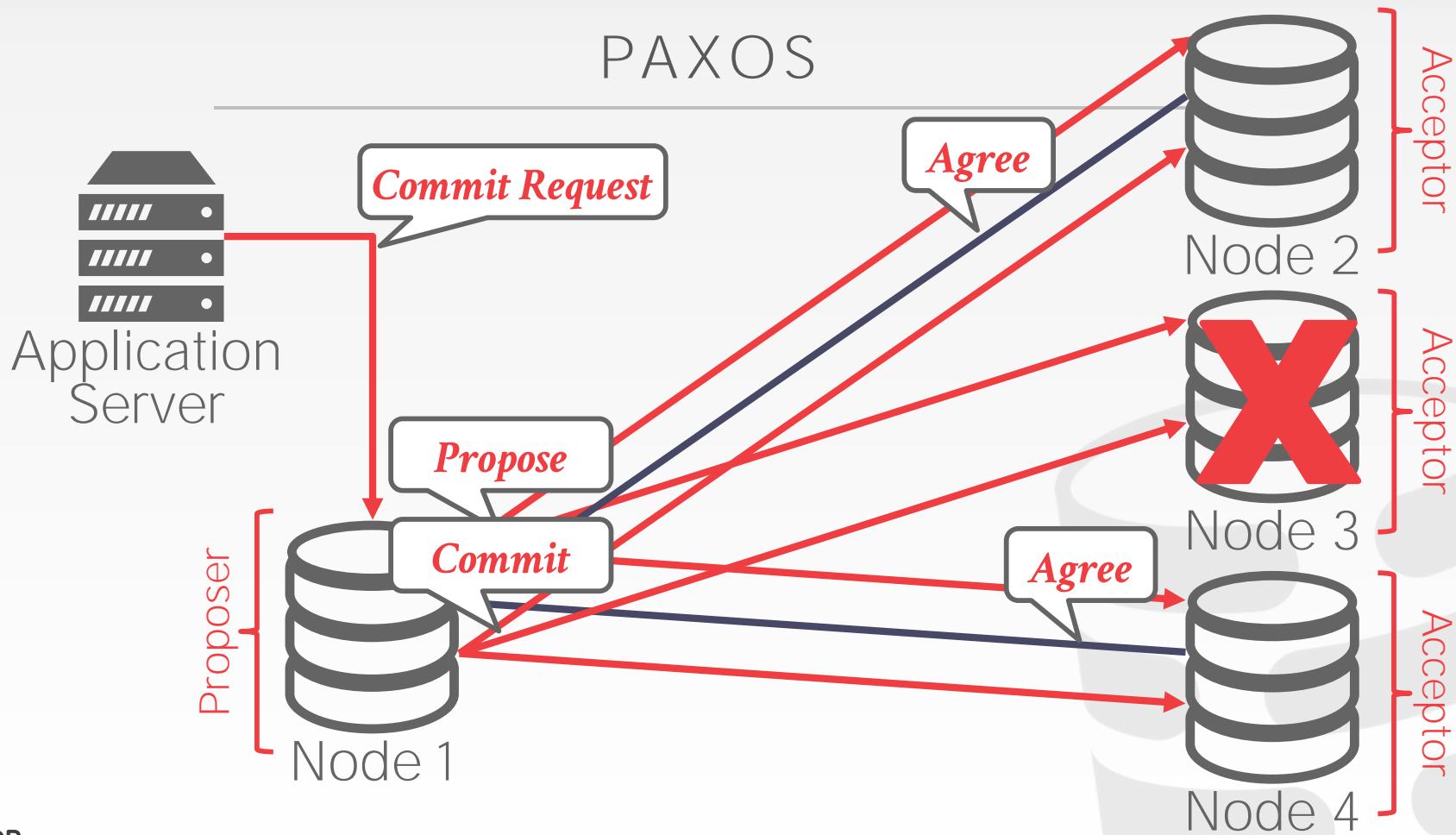
© 1998 ACM 0000-0000/98/0000-0000 \$0.00

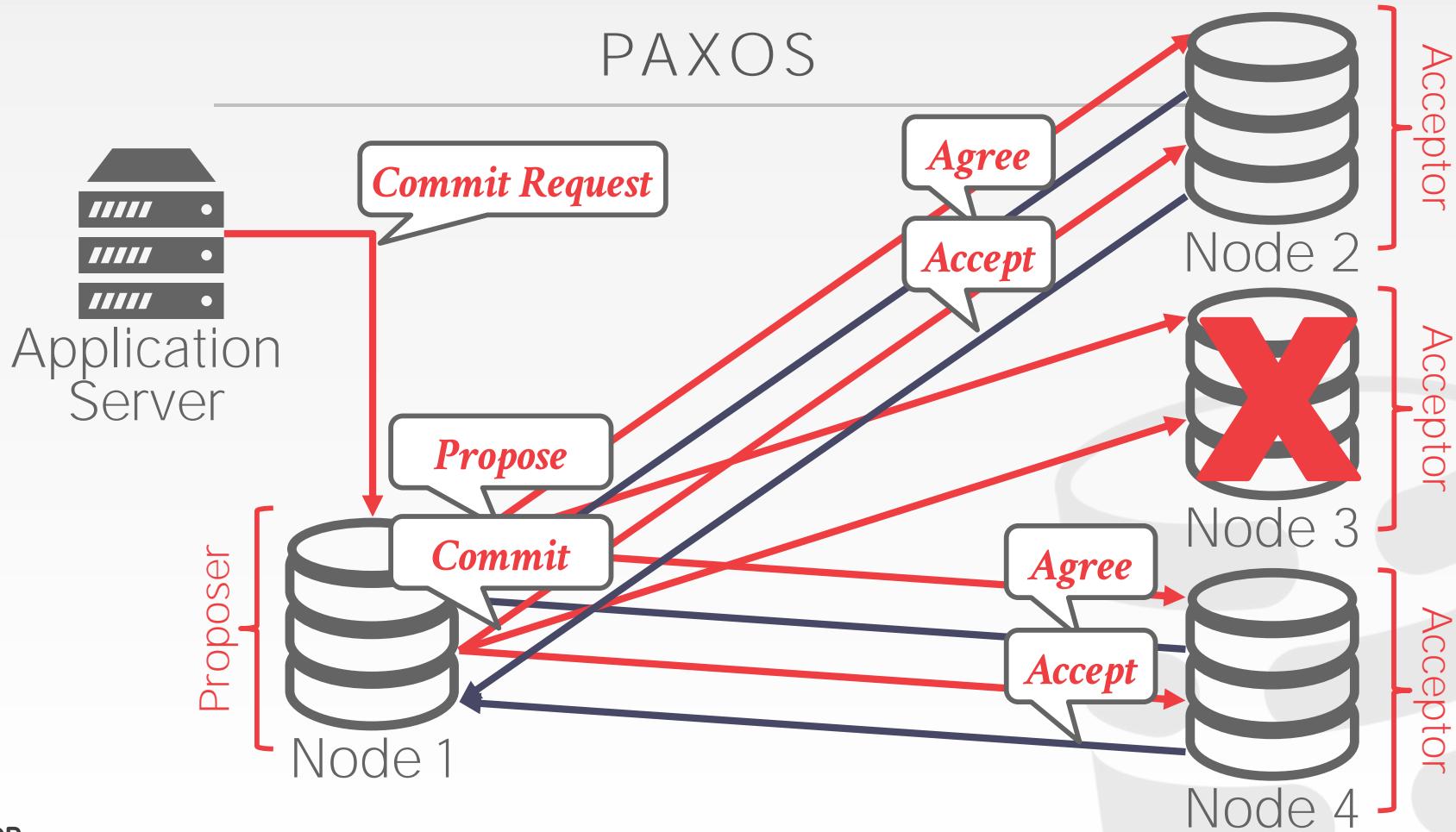


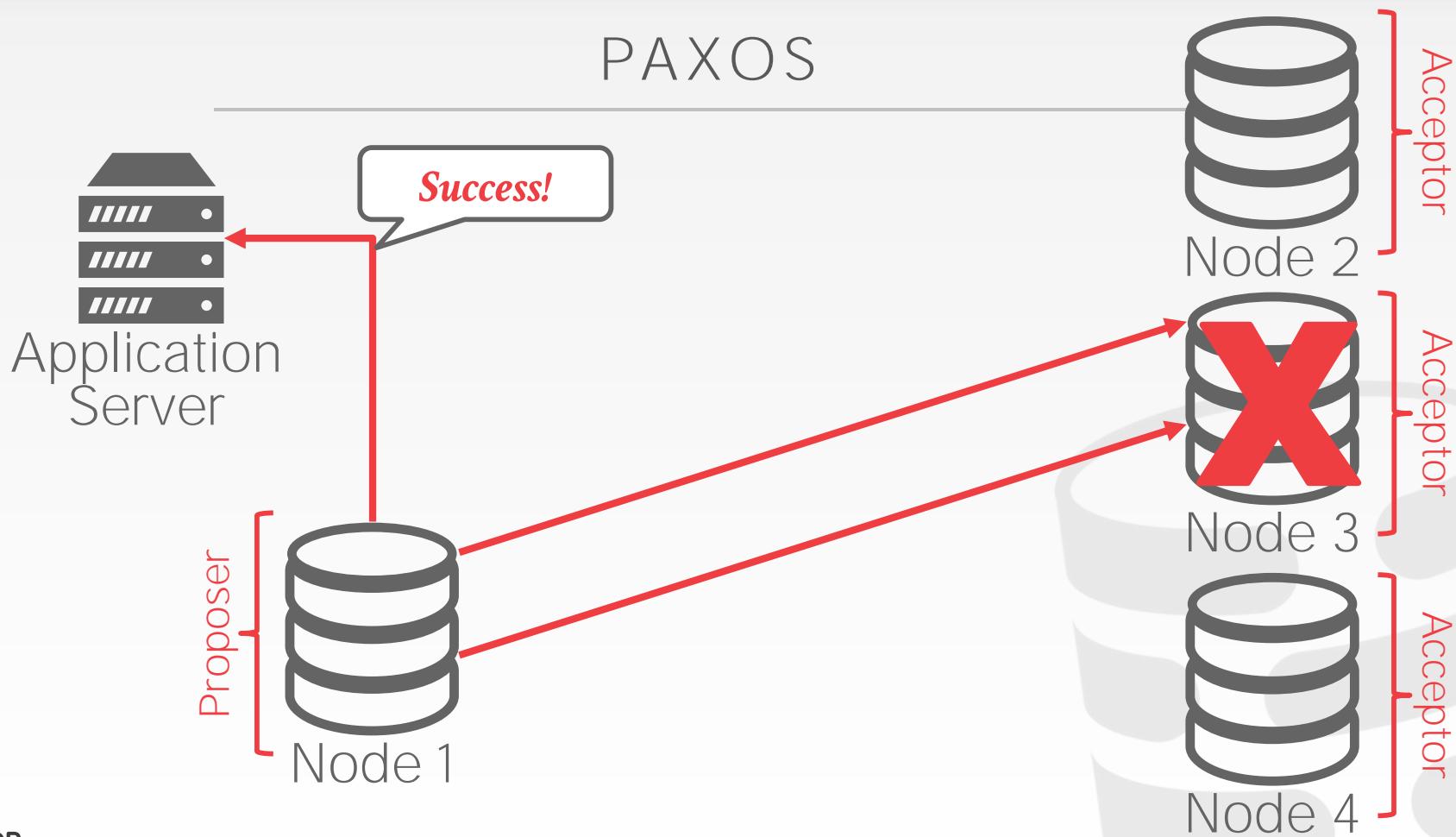




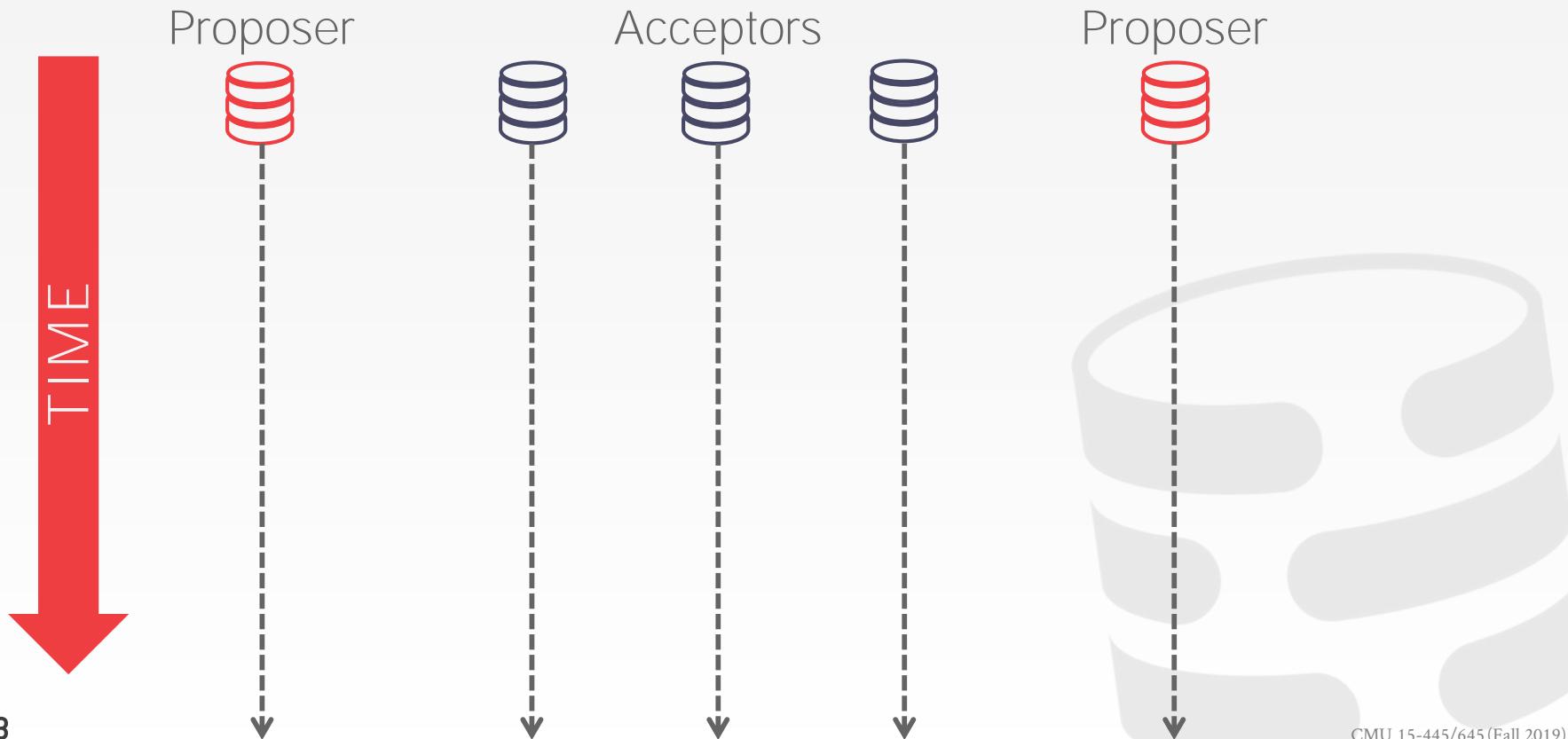




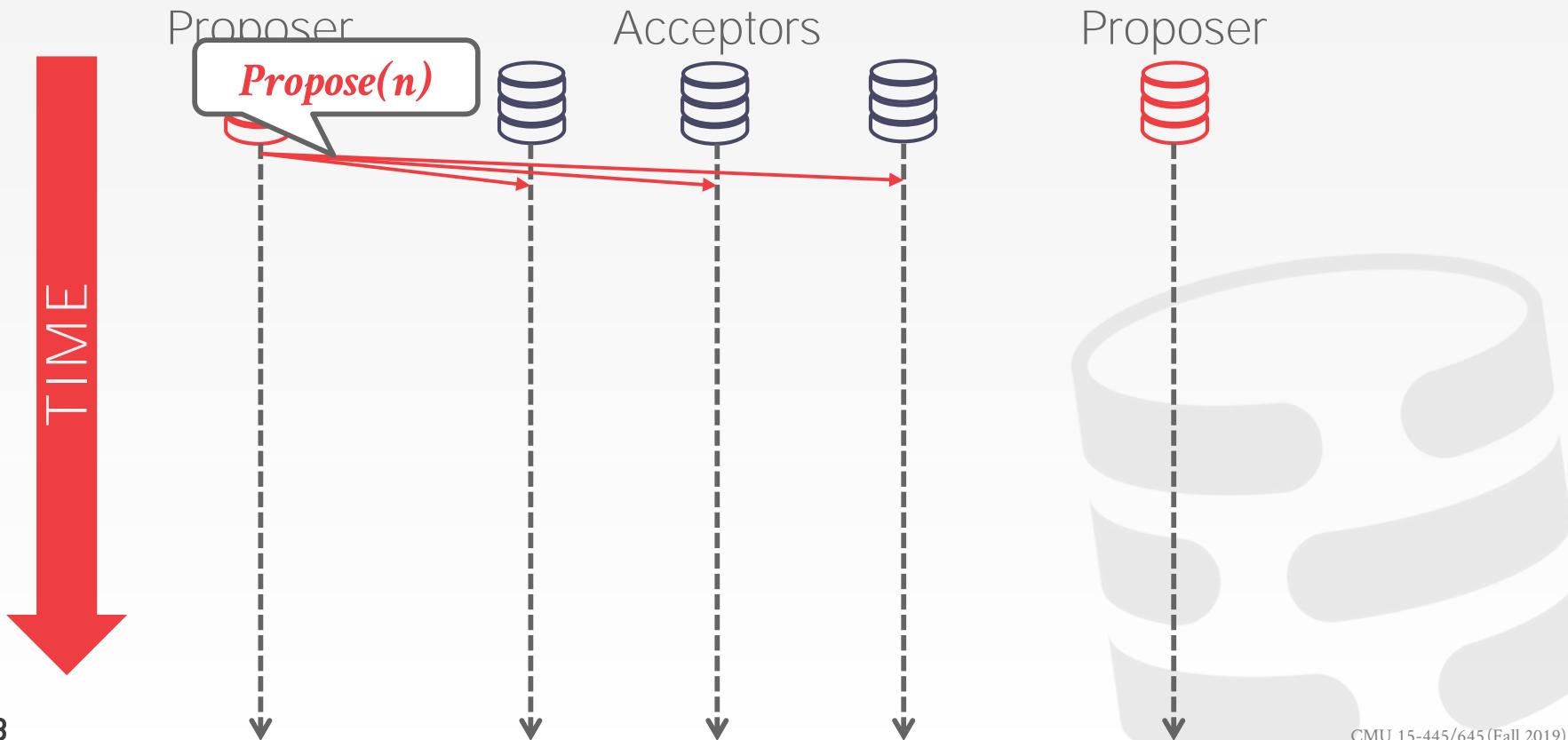




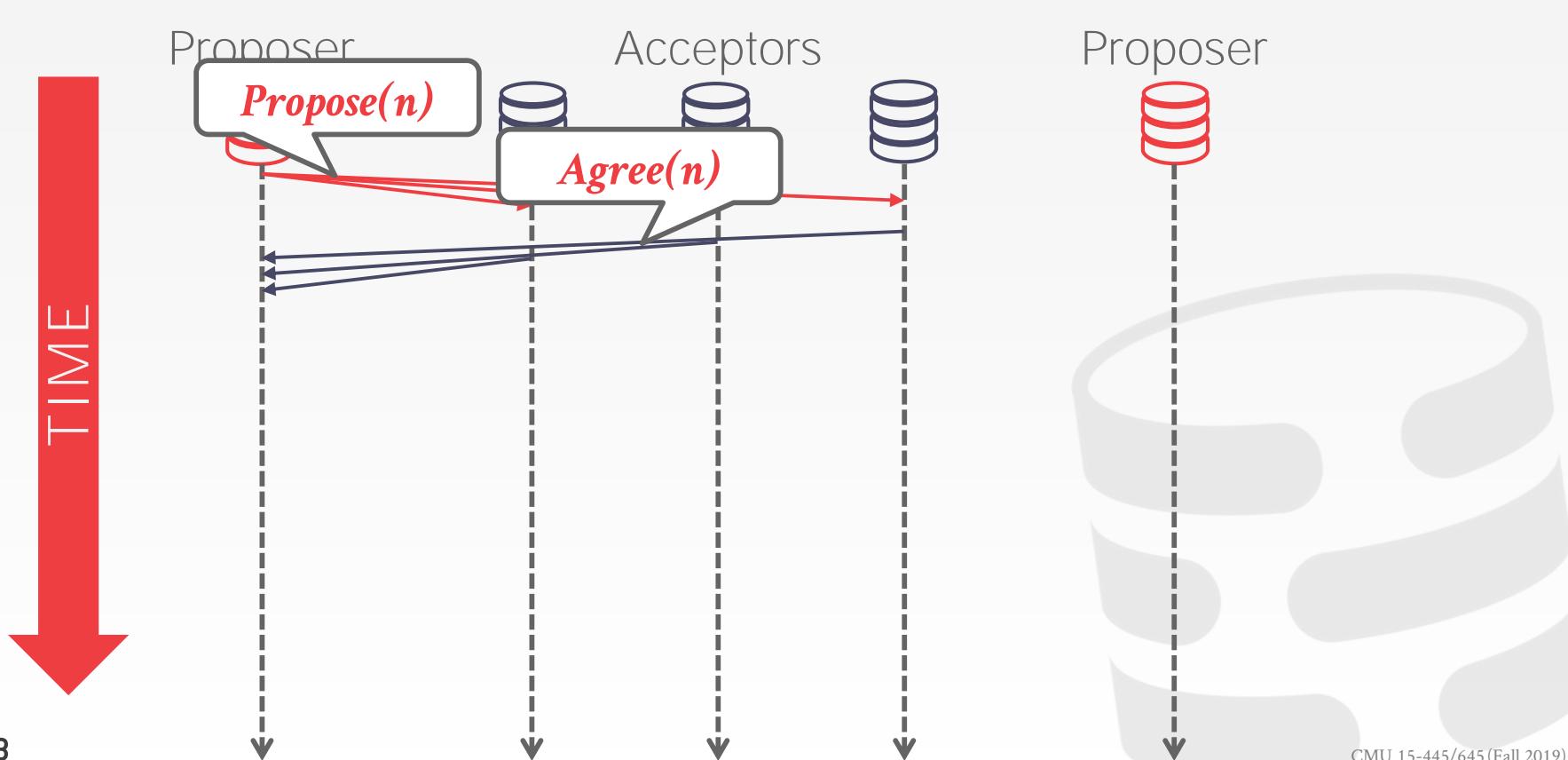
PAXOS



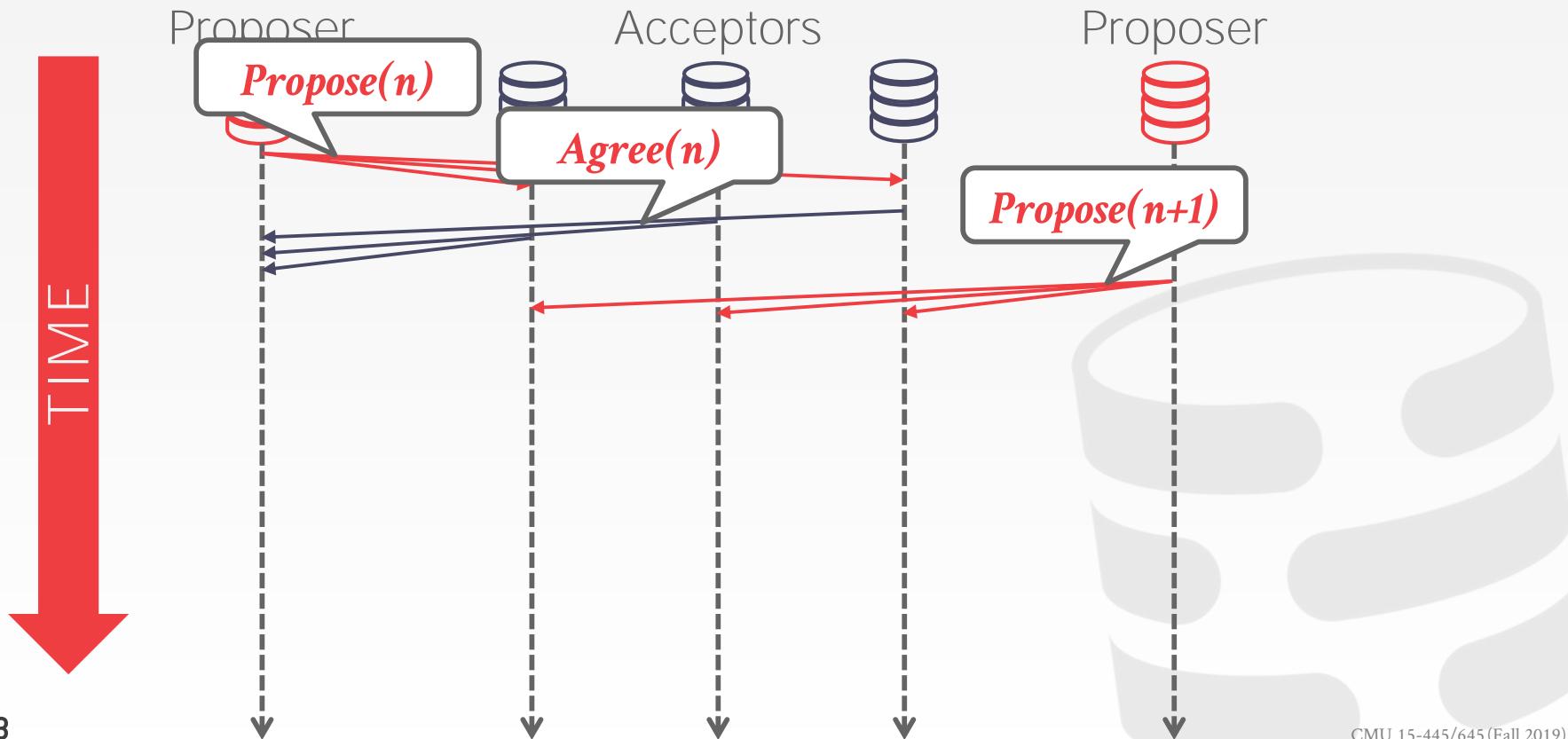
PAXOS



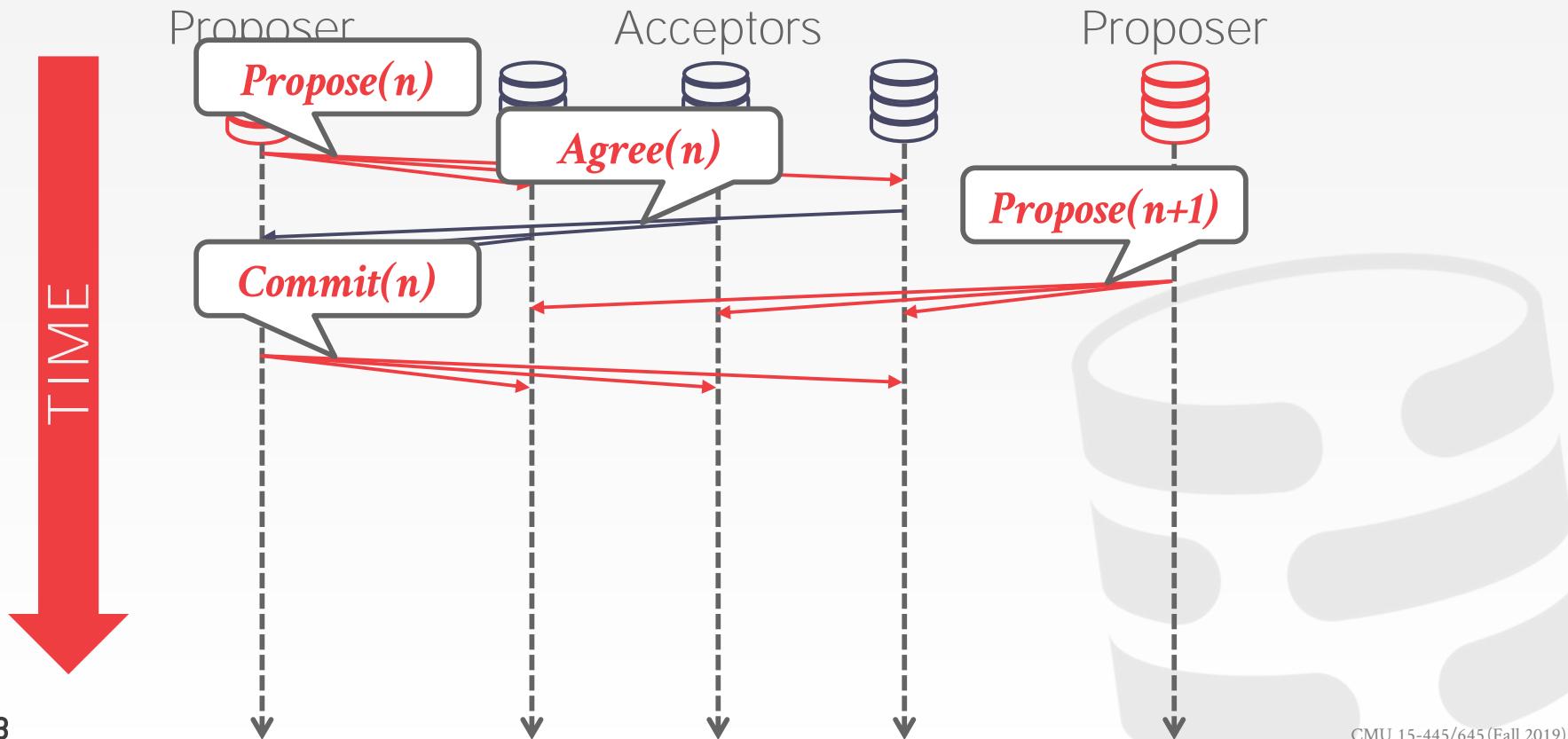
PAXOS



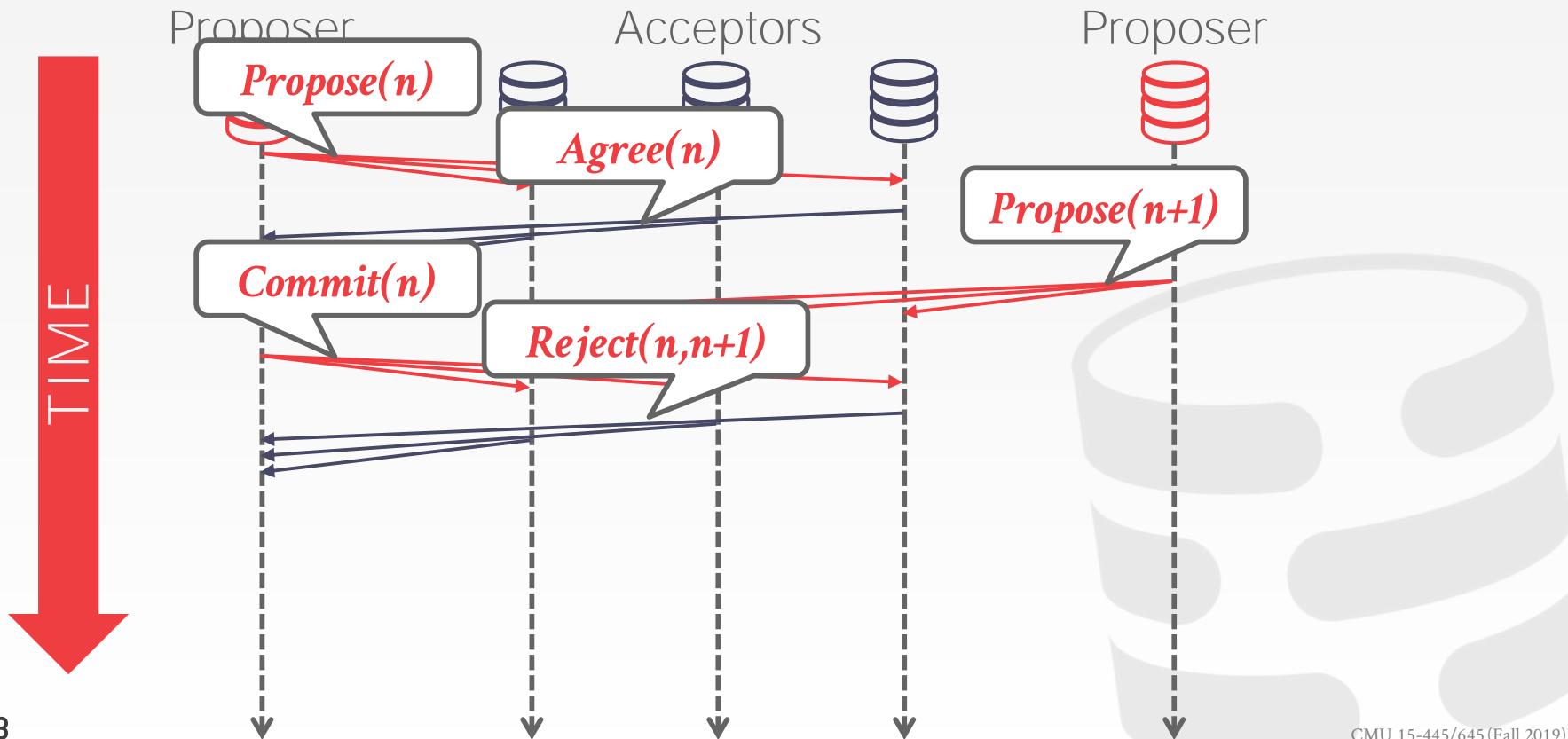
PAXOS



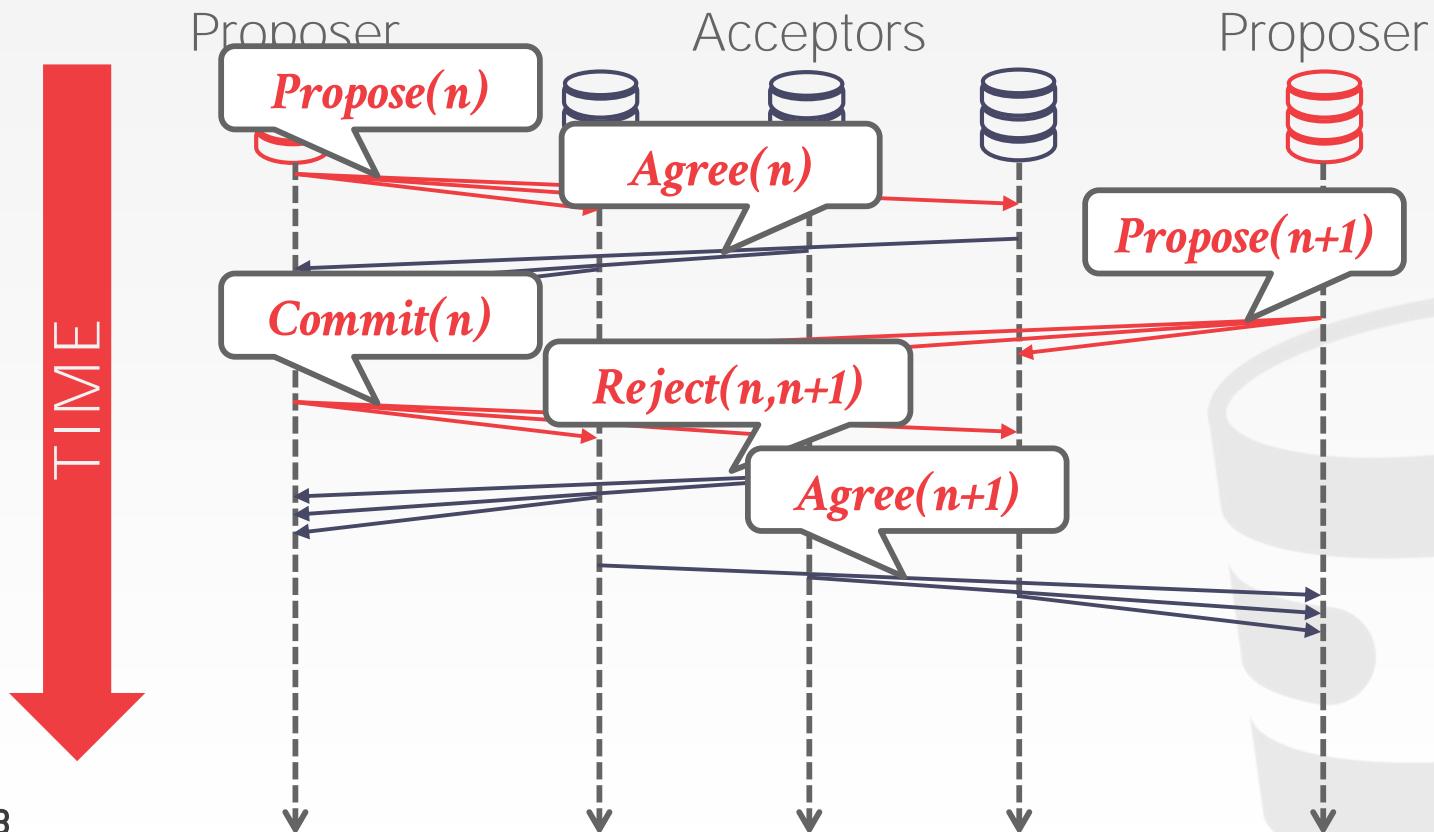
PAXOS



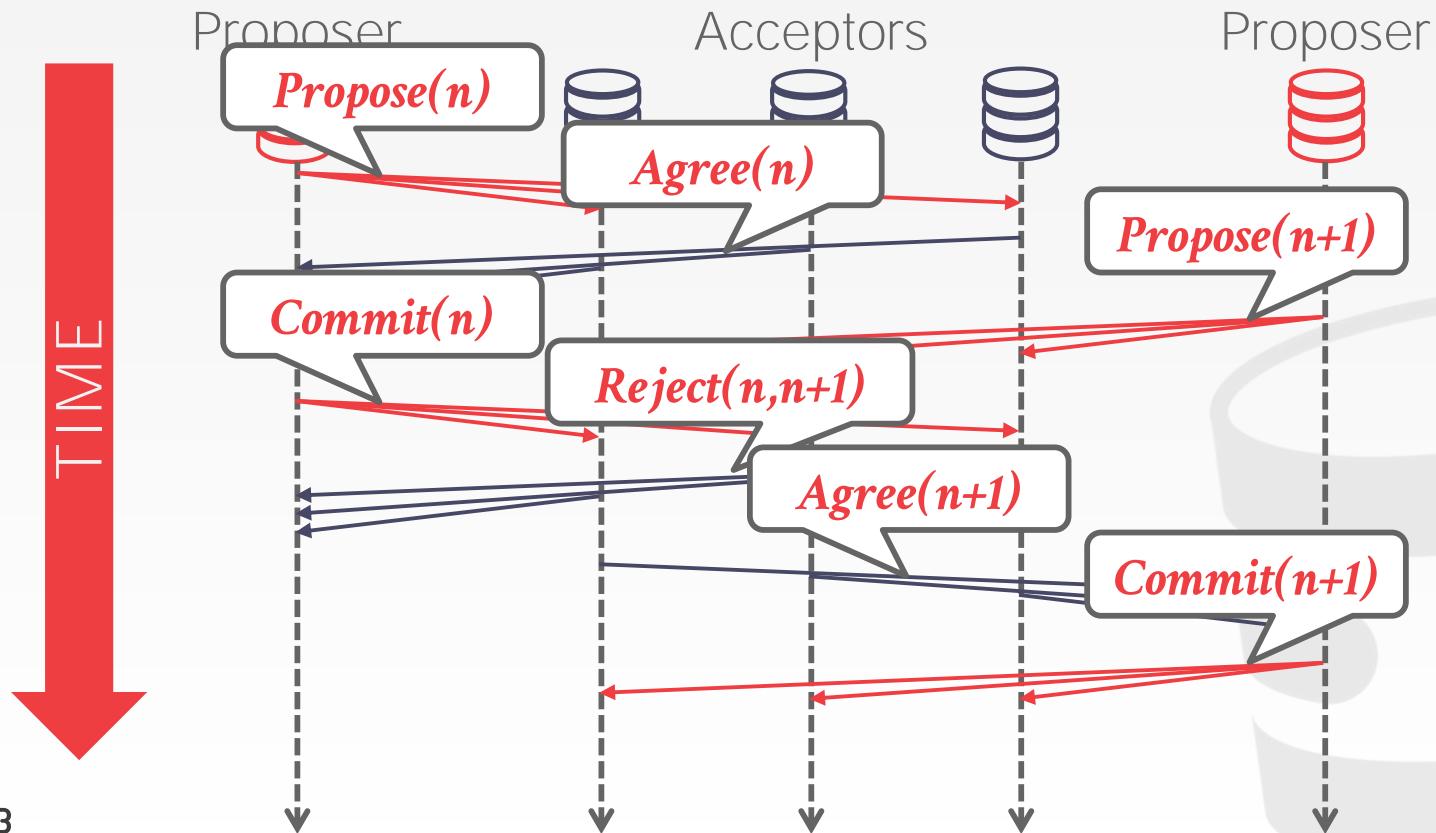
PAXOS



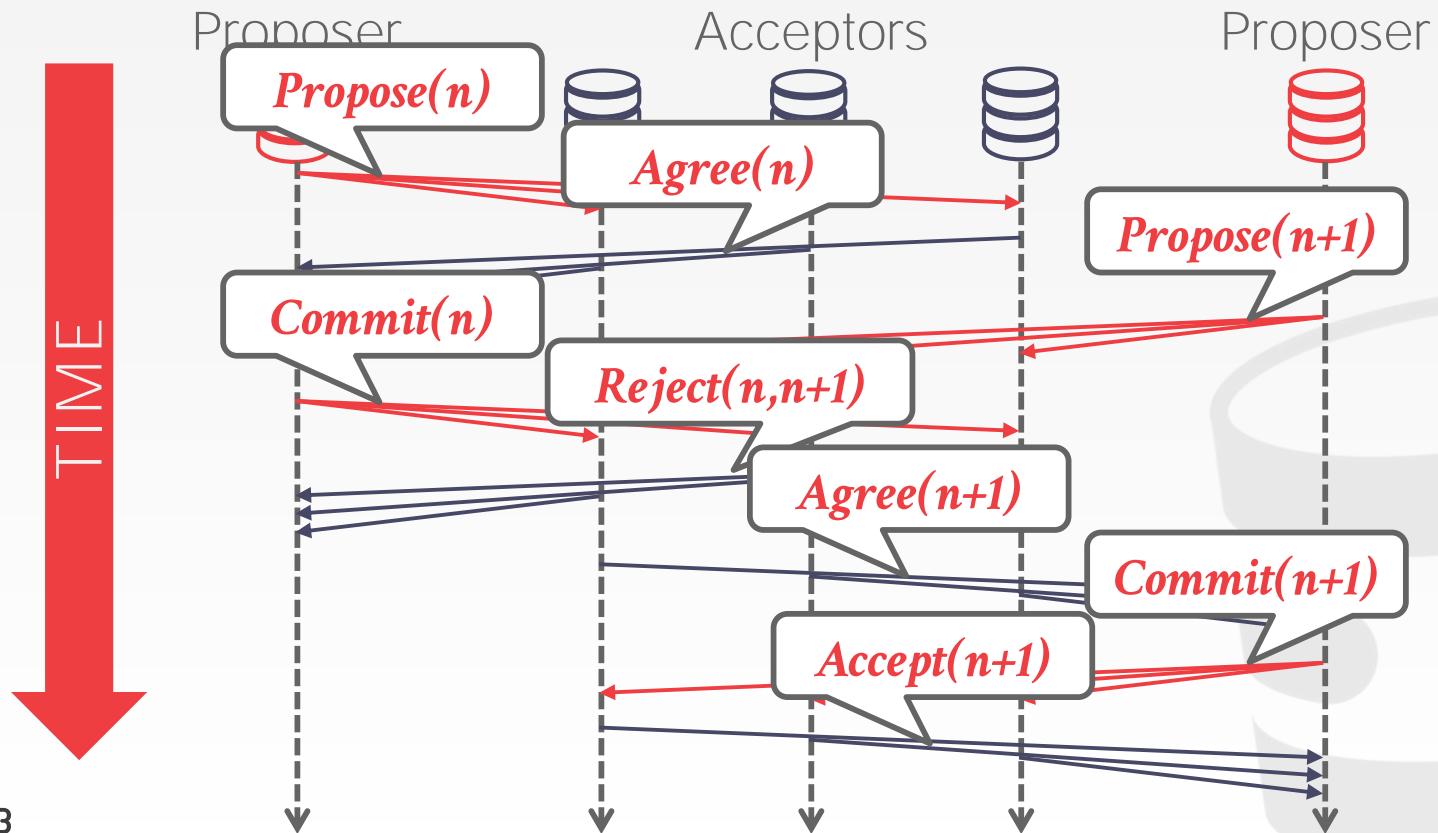
PAXOS



PAXOS



PAXOS



MULTI-PAXOS

If the system elects a single leader that is in charge of proposing changes for some period of time, then it can skip the **Propose** phase.

→ Fall back to full Paxos whenever there is a failure.

The system periodically renews who the leader is using another Paxos round.

2PC VS. PAXOS

Two-Phase Commit

- Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

Paxos

- Non-blocking if a majority participants are alive, provided there is a sufficiently long period without further failures.

REPLICATION

The DBMS can replicate data across redundant nodes to increase availability.

Design Decisions:

- Replica Configuration
- Propagation Scheme
- Propagation Timing
- Update Method



REPLICA CONFIGURATIONS

Approach #1: Master-Replica

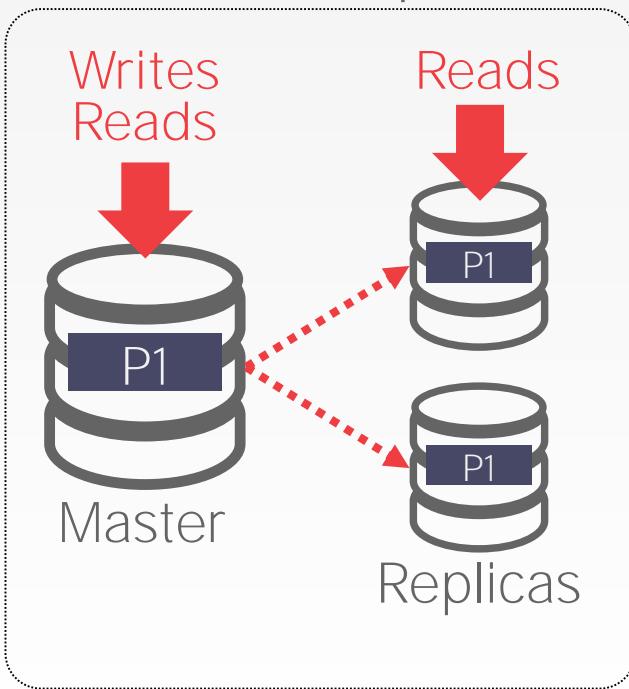
- All updates go to a designated master for each object.
- The master propagates updates to its replicas without an atomic commit protocol.
- Read-only txns may be allowed to access replicas.
- If the master goes down, then hold an election to select a new master.

Approach #2: Multi-Master

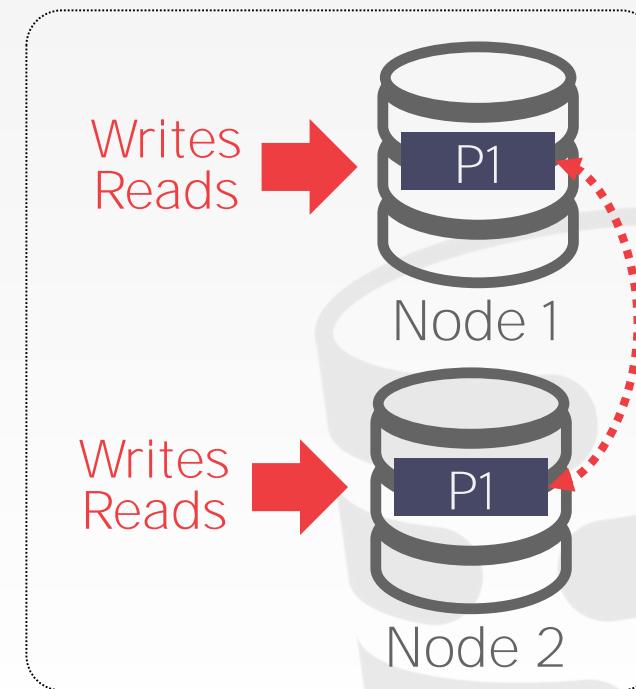
- Txns can update data objects at any replica.
- Replicas must synchronize with each other using an atomic commit protocol.

REPLICA CONFIGURATIONS

Master-Replica



Multi-Master



K-SAFETY

K-safety is a threshold for determining the fault tolerance of the replicated database.

The value *K* represents the number of replicas per data object that must always be available.

If the number of replicas goes below this threshold, then the DBMS halts execution and takes itself offline.



PROPAGATION SCHEME

When a txn commits on a replicated database, the DBMS decides whether it must wait for that txn's changes to propagate to other nodes before it can send the acknowledgement to application.

Propagation levels:

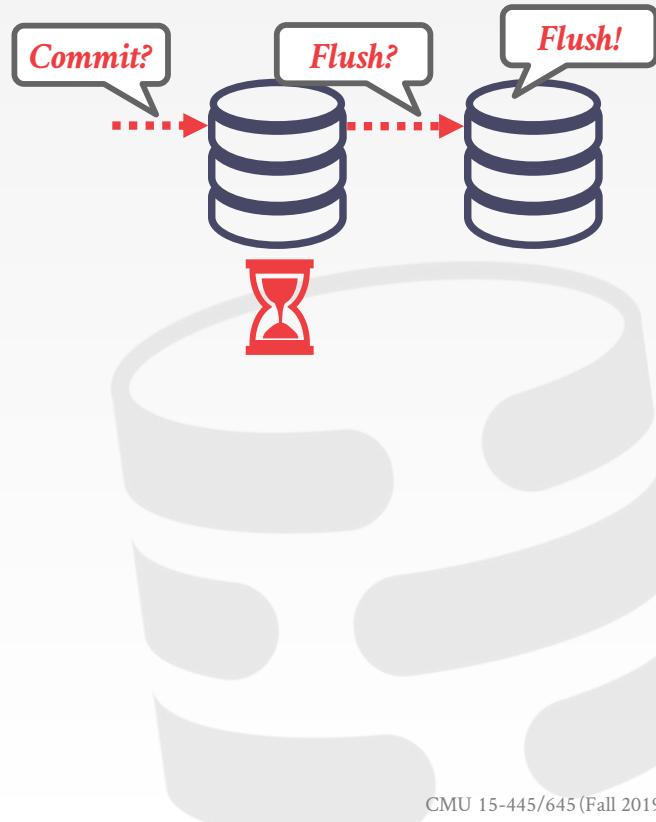
- Synchronous (*Strong Consistency*)
- Asynchronous (*Eventual Consistency*)



PROPAGATION SCHEME

Approach #1: Synchronous

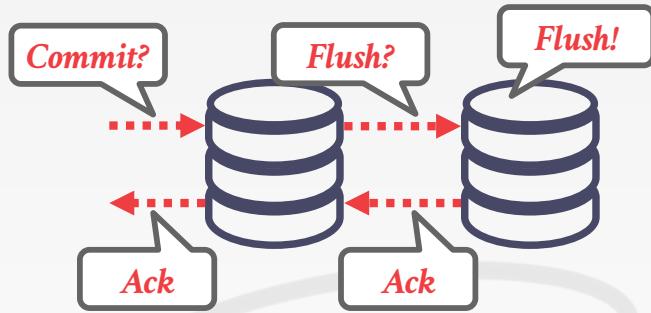
- The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



PROPAGATION SCHEME

Approach #1: Synchronous

- The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



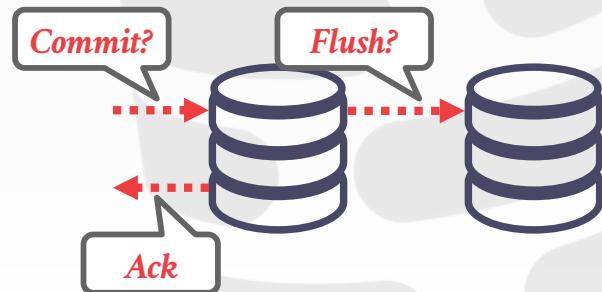
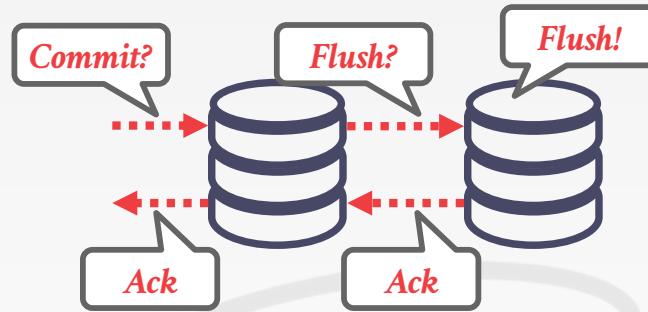
PROPAGATION SCHEME

Approach #1: Synchronous

- The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

Approach #2: Asynchronous

- The master immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.



PROPAGATION TIMING

Approach #1: Continuous

- The DBMS sends log messages immediately as it generates them.
- Also need to send a commit/abort message.

Approach #2: On Commit

- The DBMS only sends the log messages for a txn to the replicas once the txn is commits.
- Do not waste time sending log records for aborted txns.
- Assumes that a txn's log records fits entirely in memory.

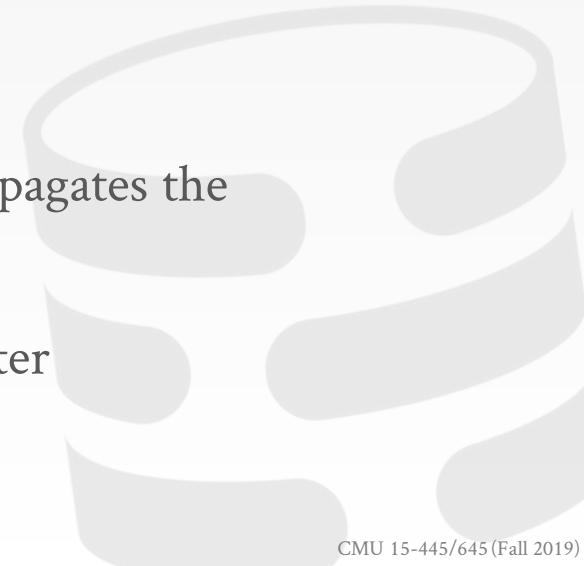
ACTIVE VS. PASSIVE

Approach #1: Active-Active

- A txn executes at each replica independently.
- Need to check at the end whether the txn ends up with the same result at each replica.

Approach #2: Active-Passive

- Each txn executes at a single location and propagates the changes to the replica.
- Can either do physical or logical replication.
- Not the same as master-replica vs. multi-master



CAP THEOREM

Proposed by Eric Brewer that it is impossible for a distributed system to always be:

- Consistent
- Always Available
- Network Partition Tolerant

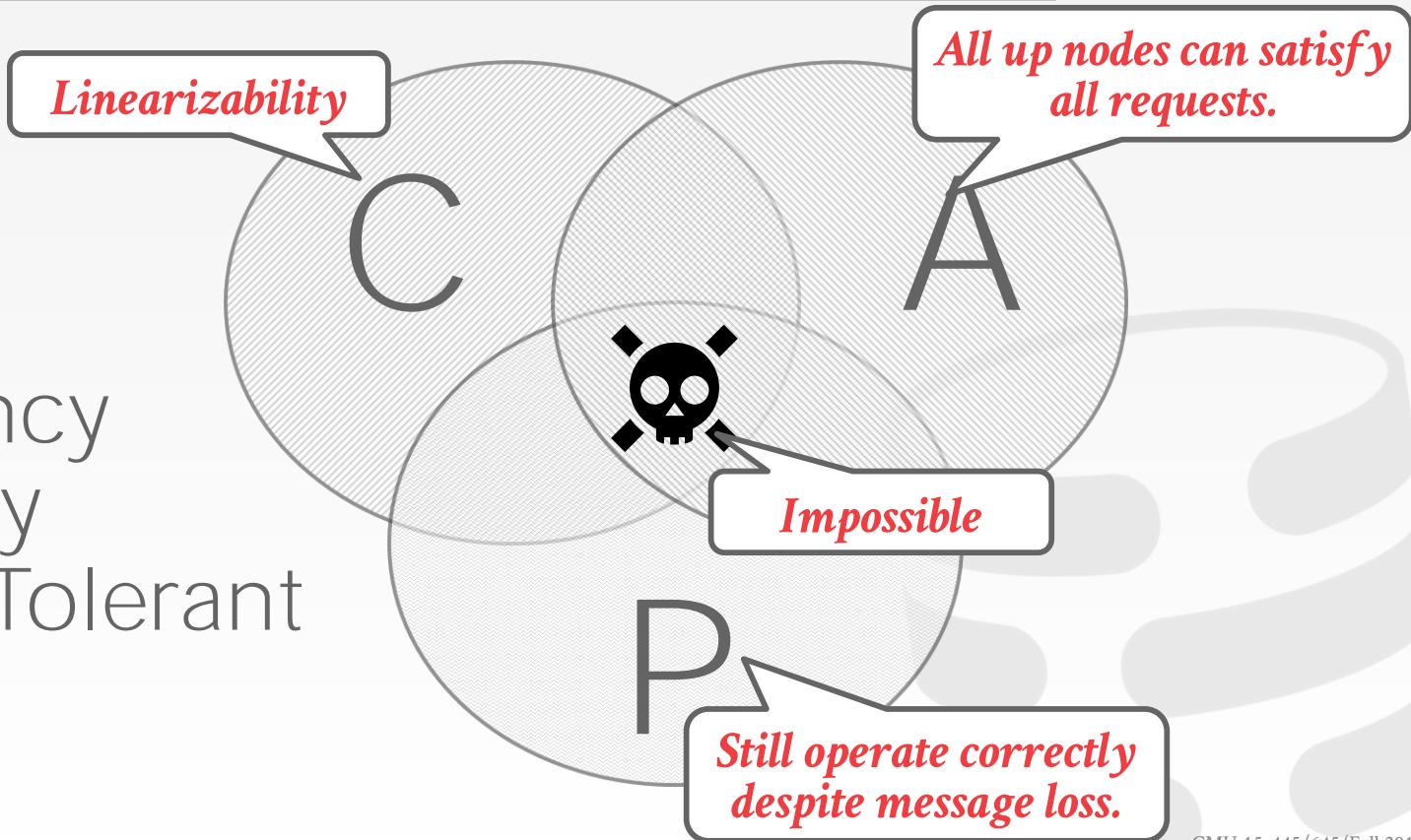
Proved in 2002.



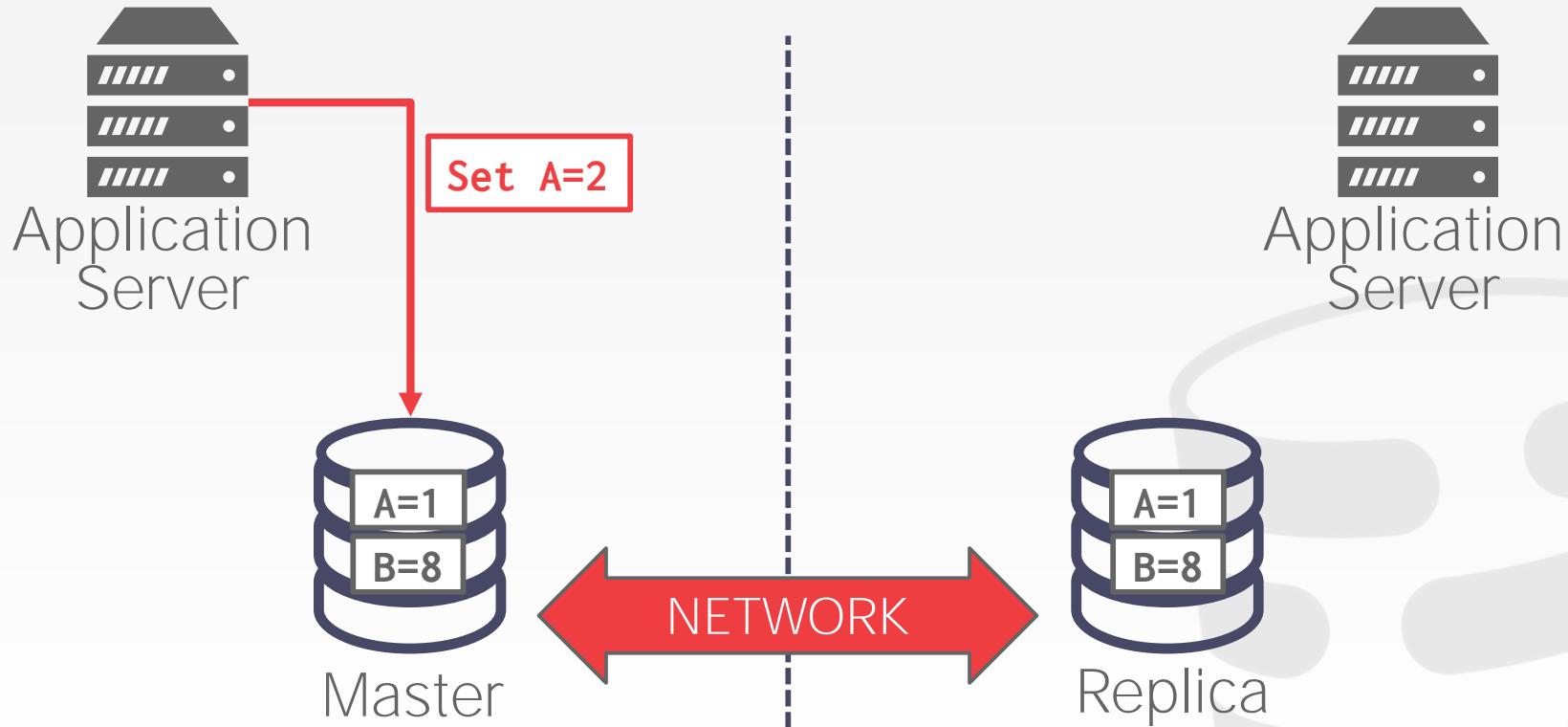
Brewer

CAP THEOREM

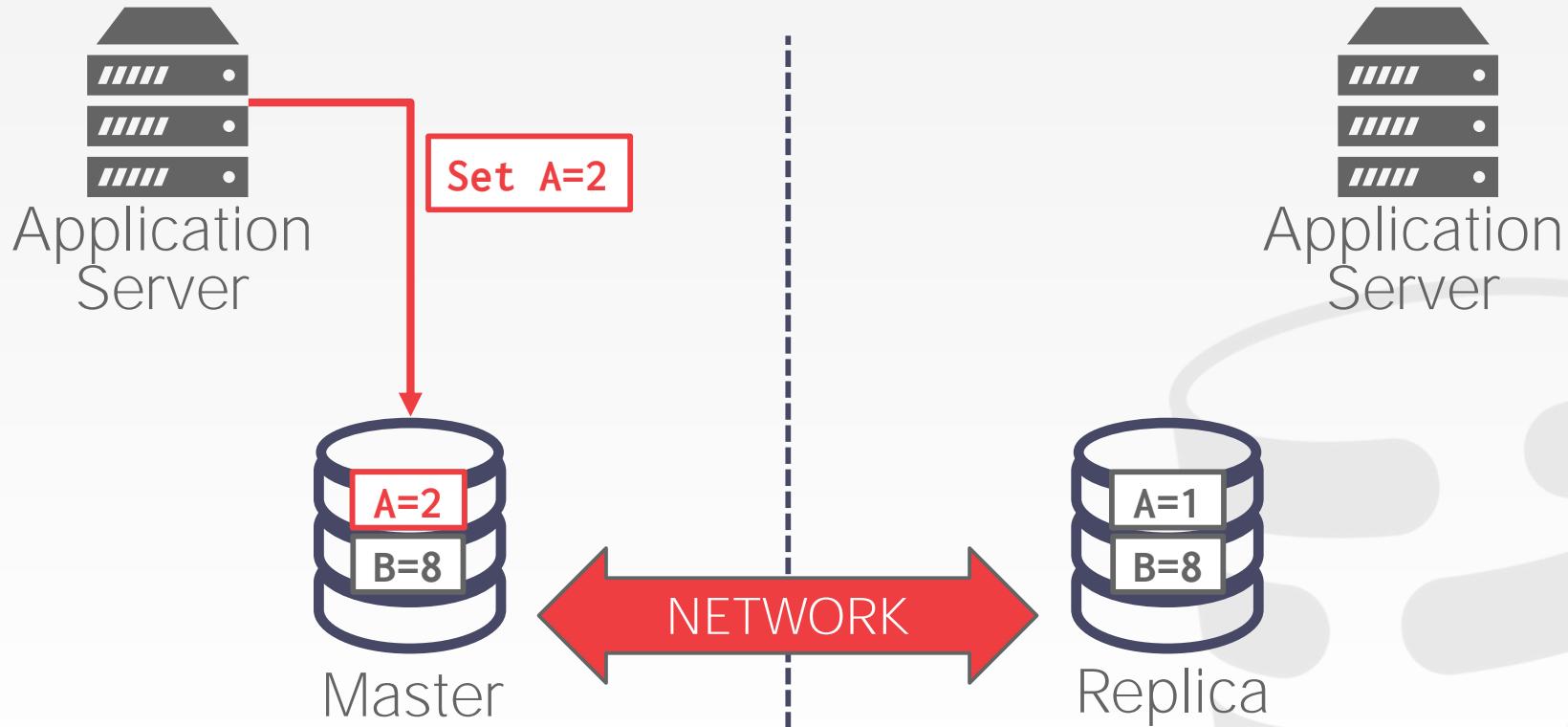
Consistency
Availability
Partition Tolerant



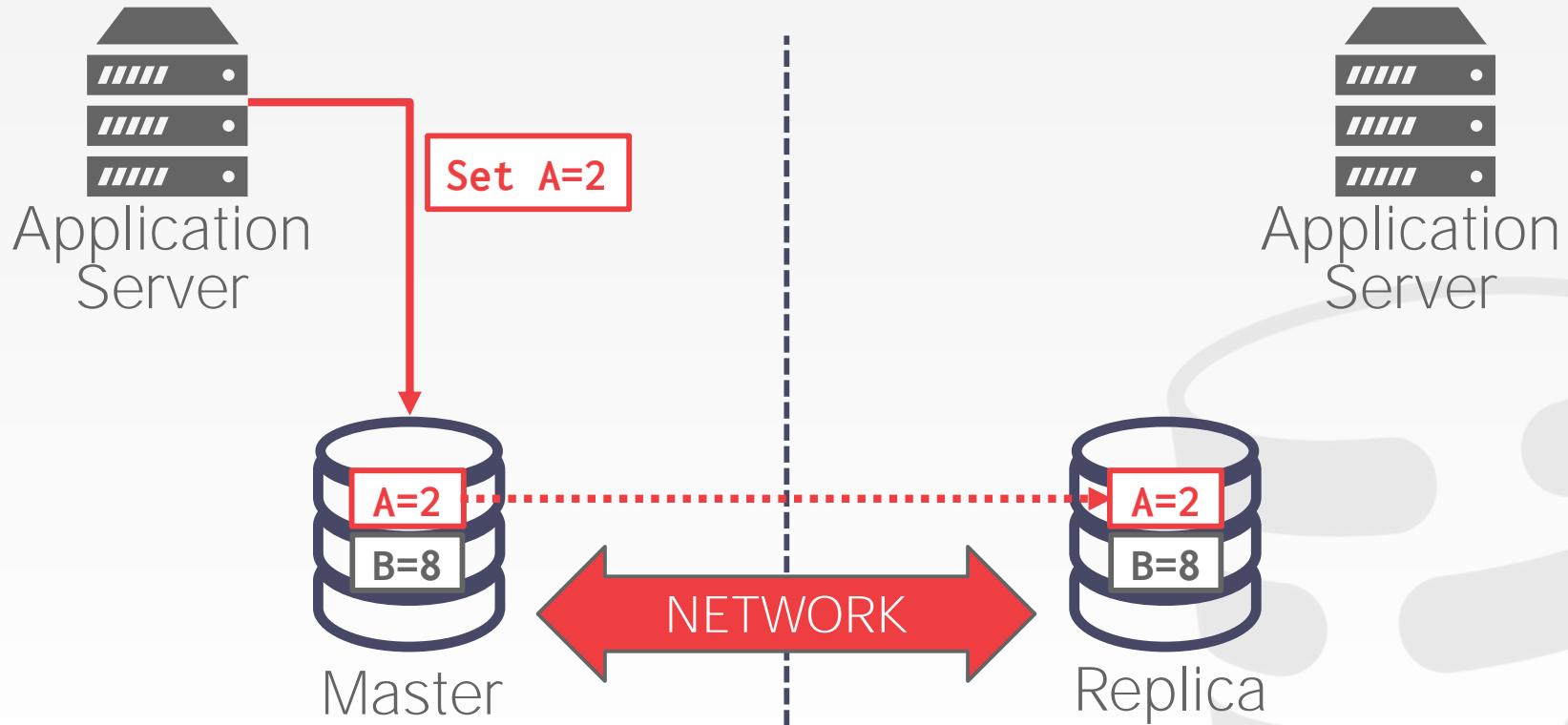
CAP – CONSISTENCY



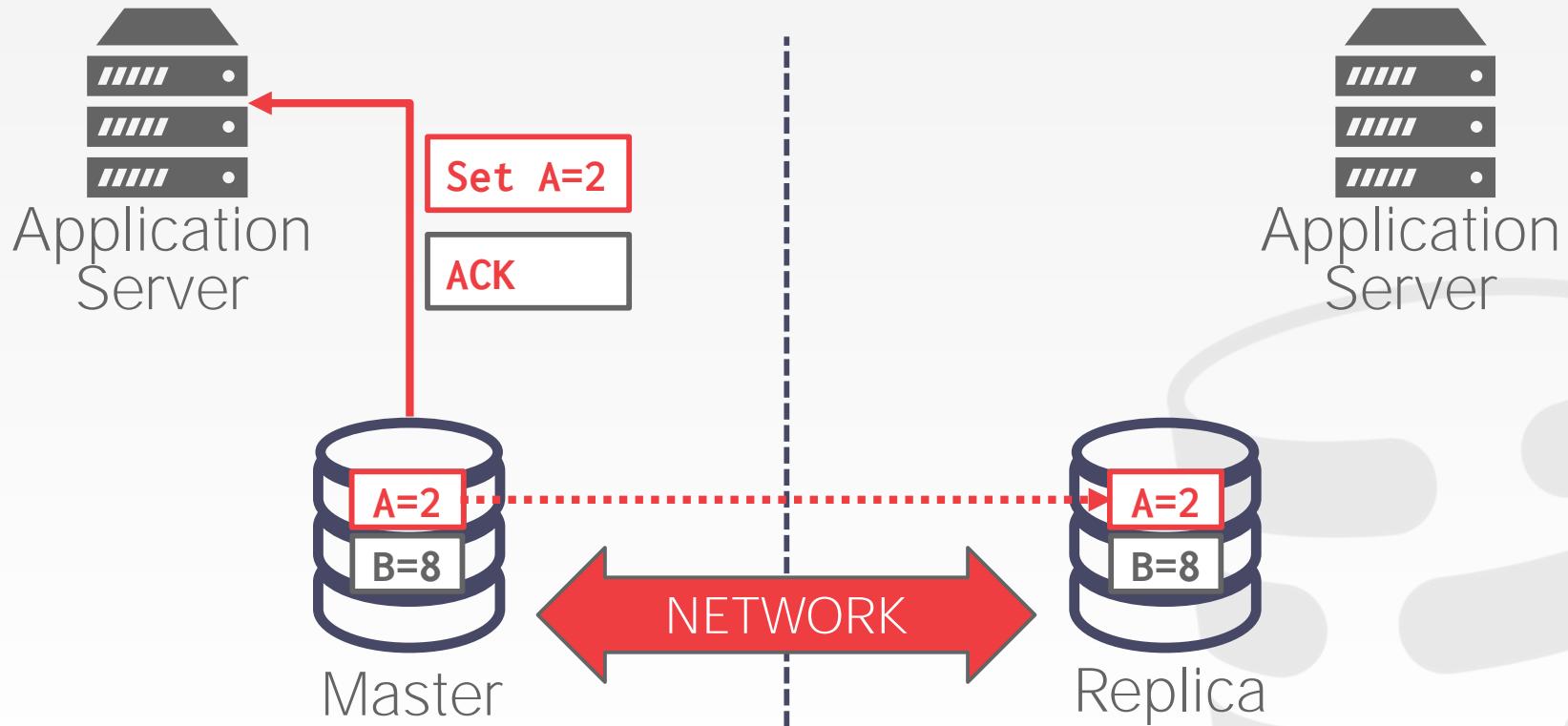
CAP – CONSISTENCY



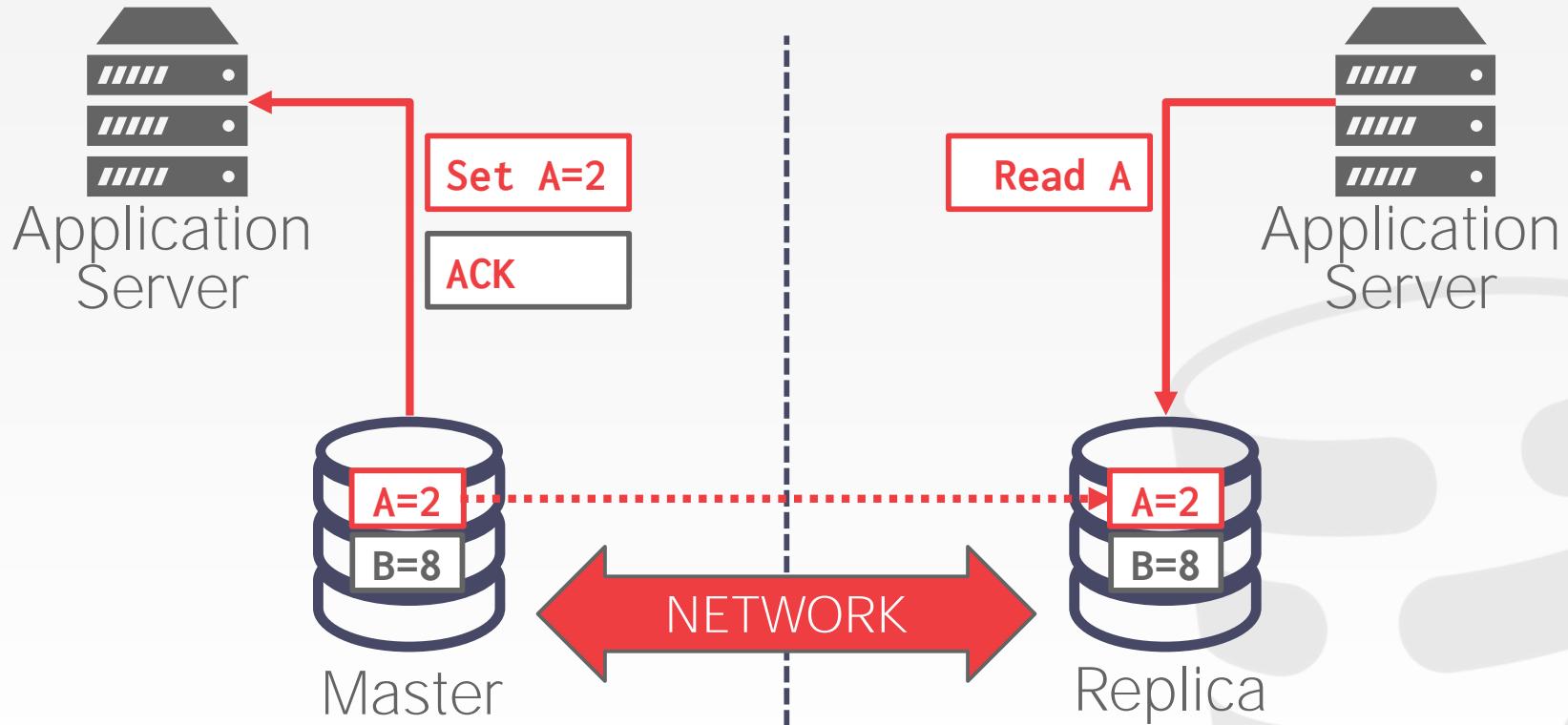
CAP – CONSISTENCY

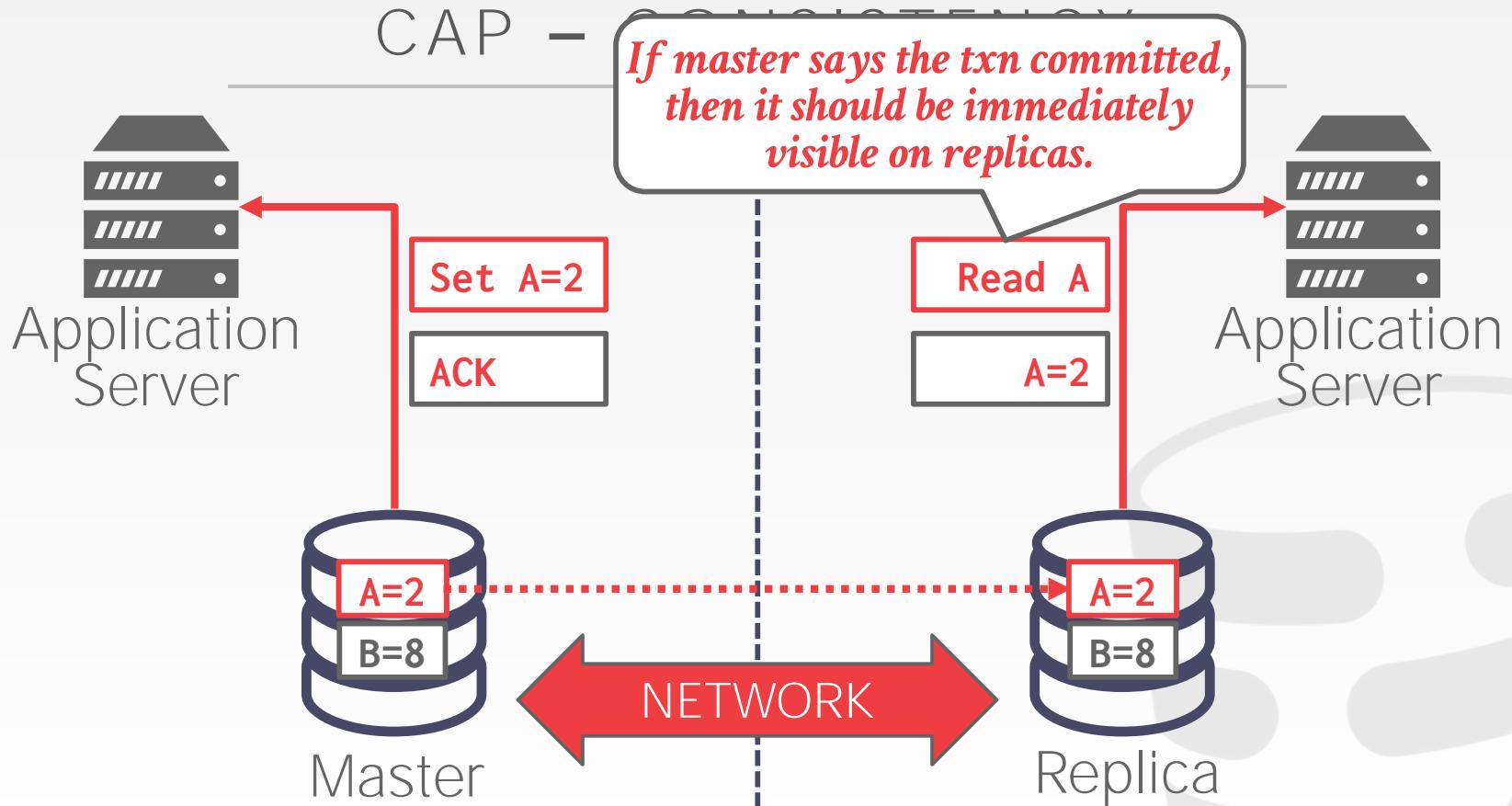


CAP – CONSISTENCY

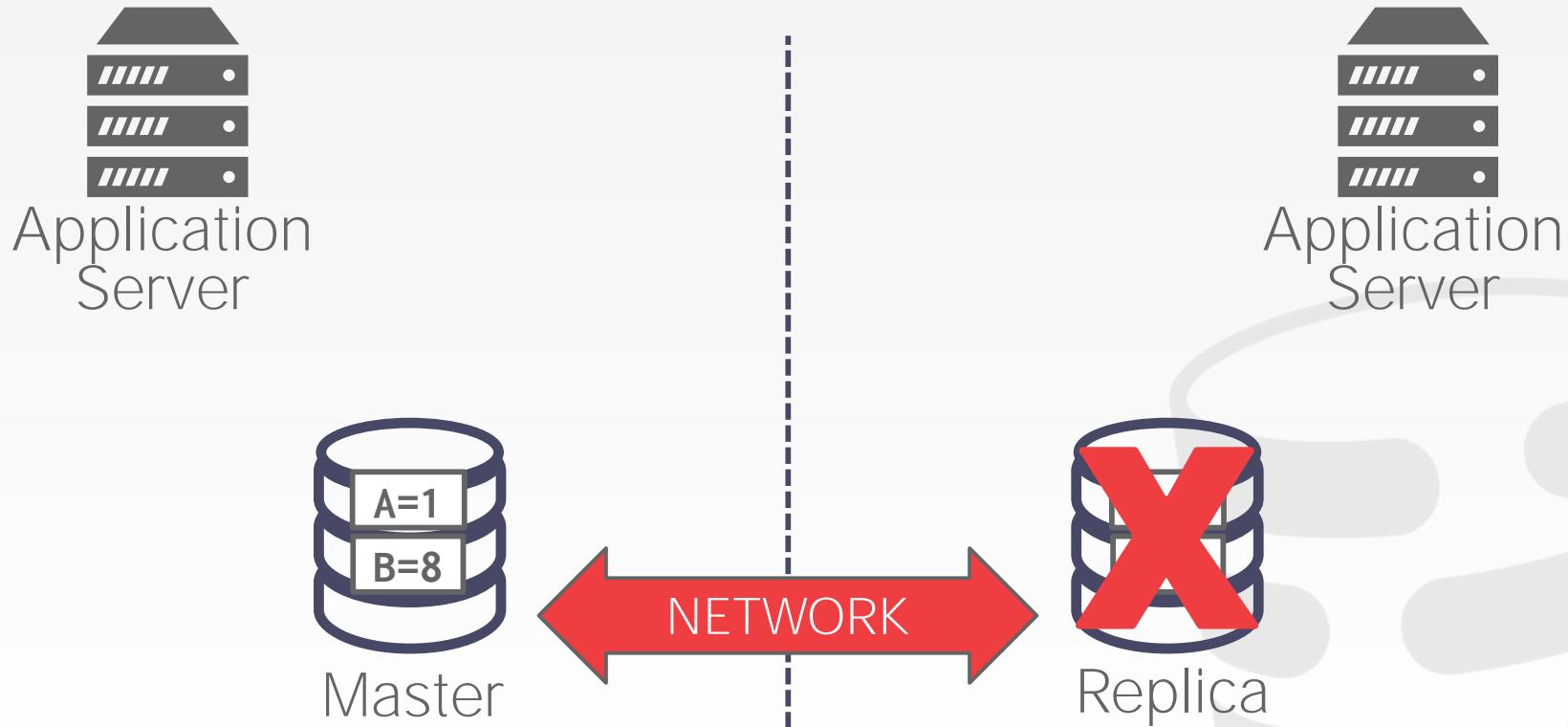


CAP – CONSISTENCY

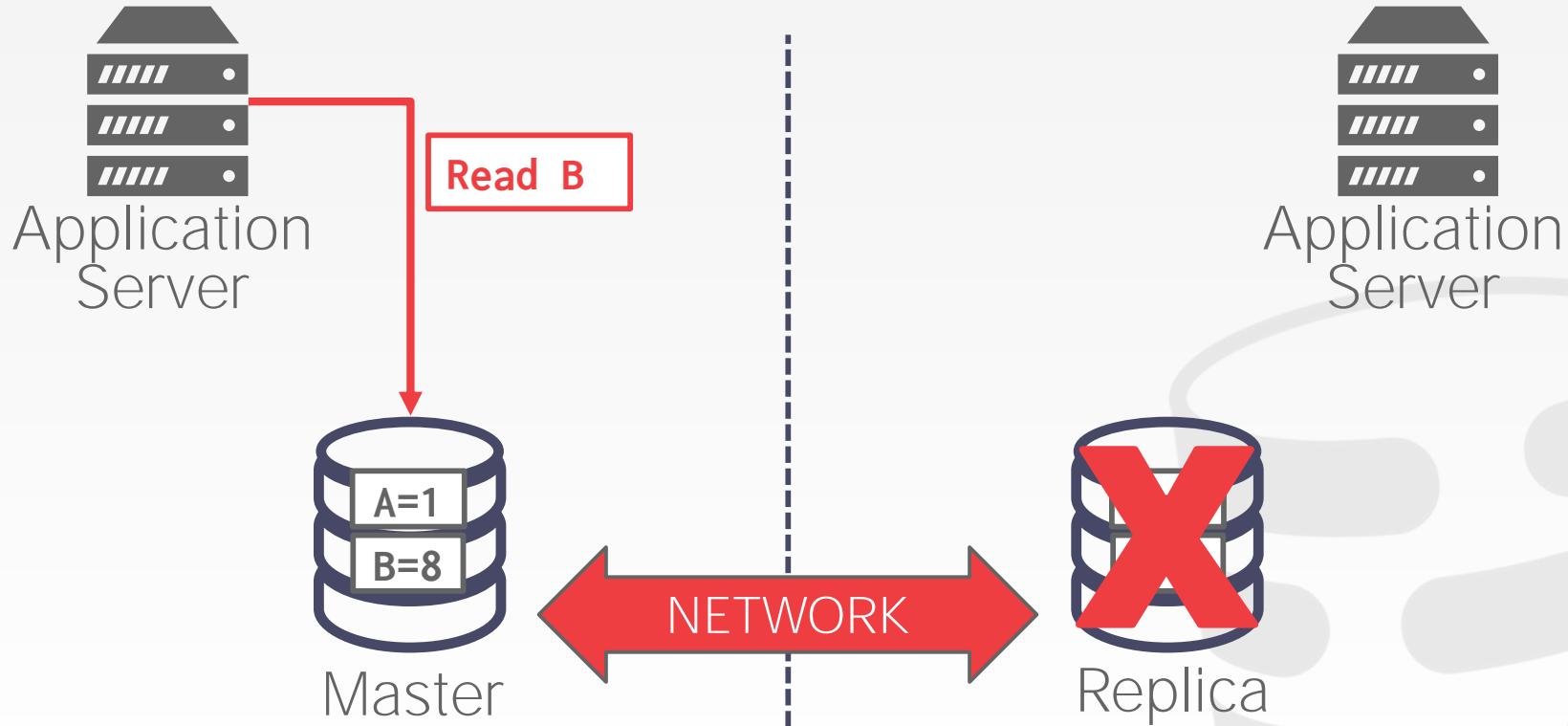




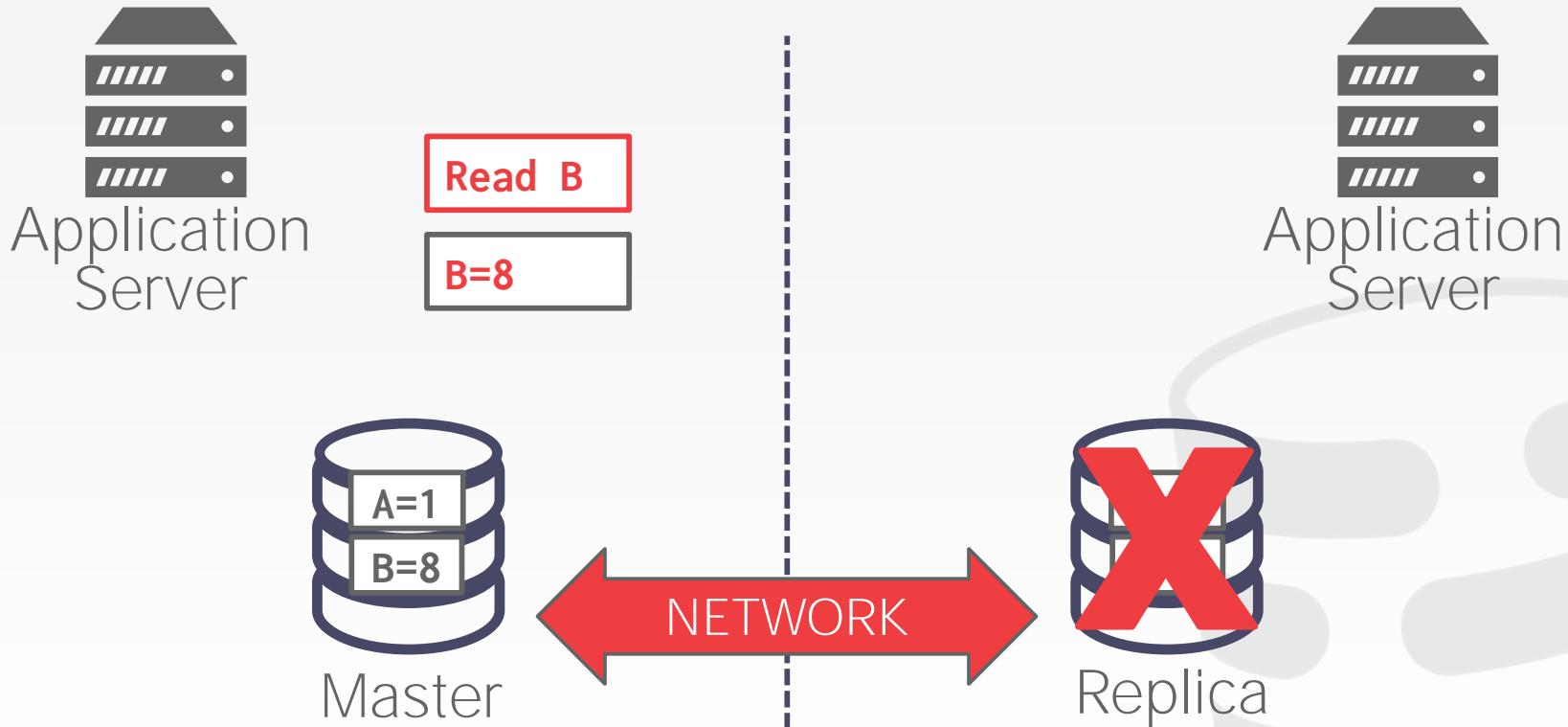
CAP – AVAILABILITY



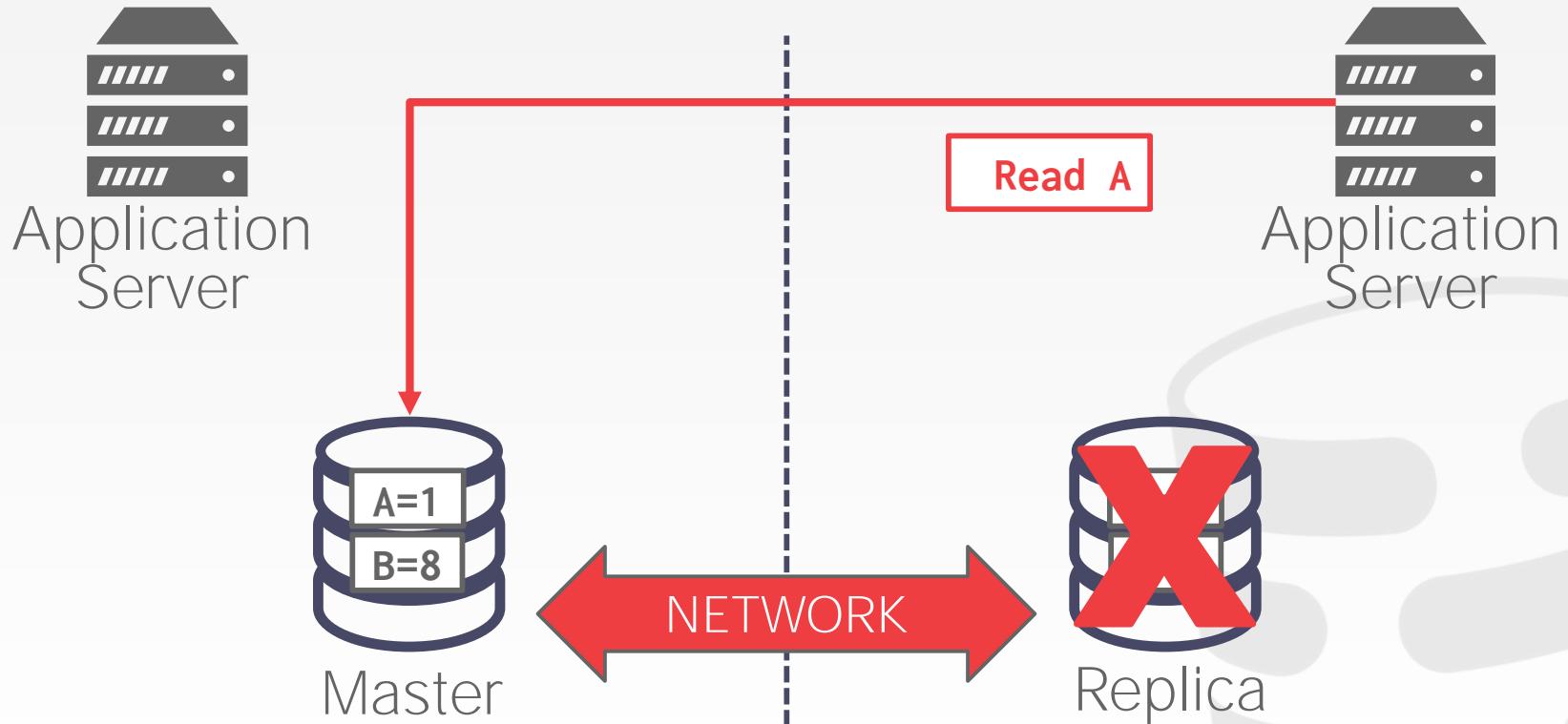
CAP – AVAILABILITY



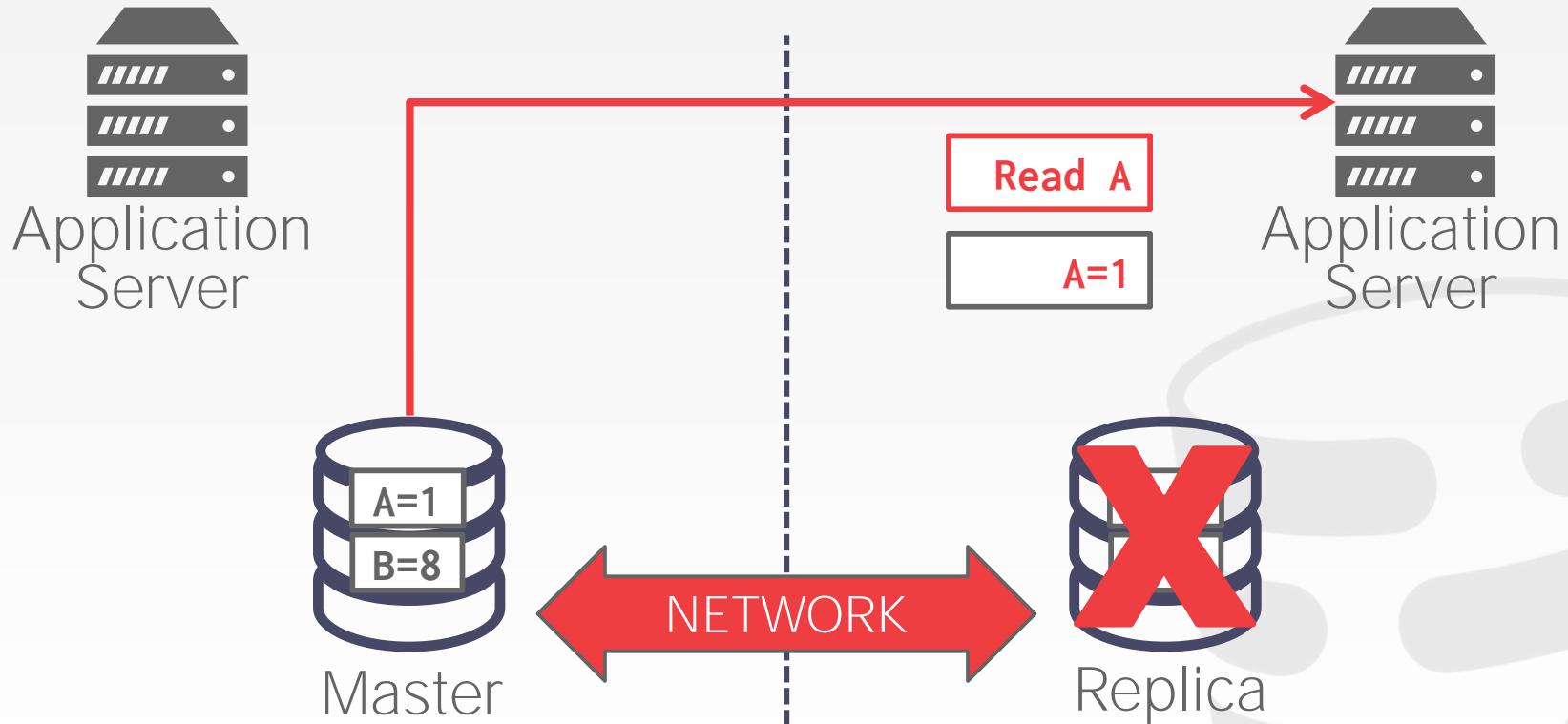
CAP – AVAILABILITY



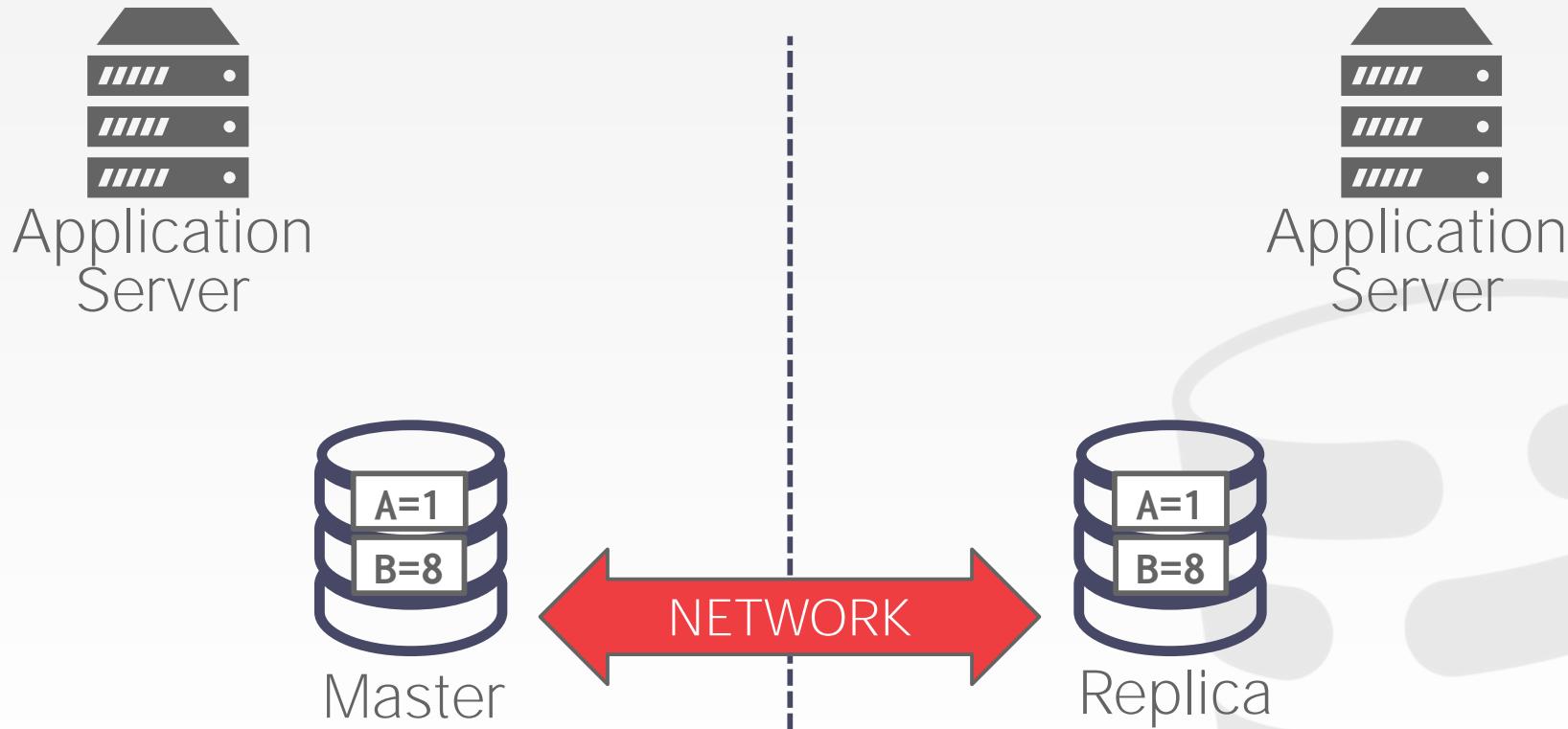
CAP – AVAILABILITY



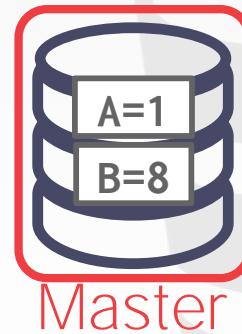
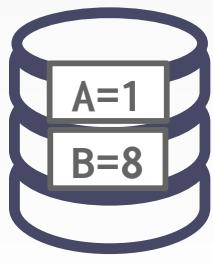
CAP – AVAILABILITY



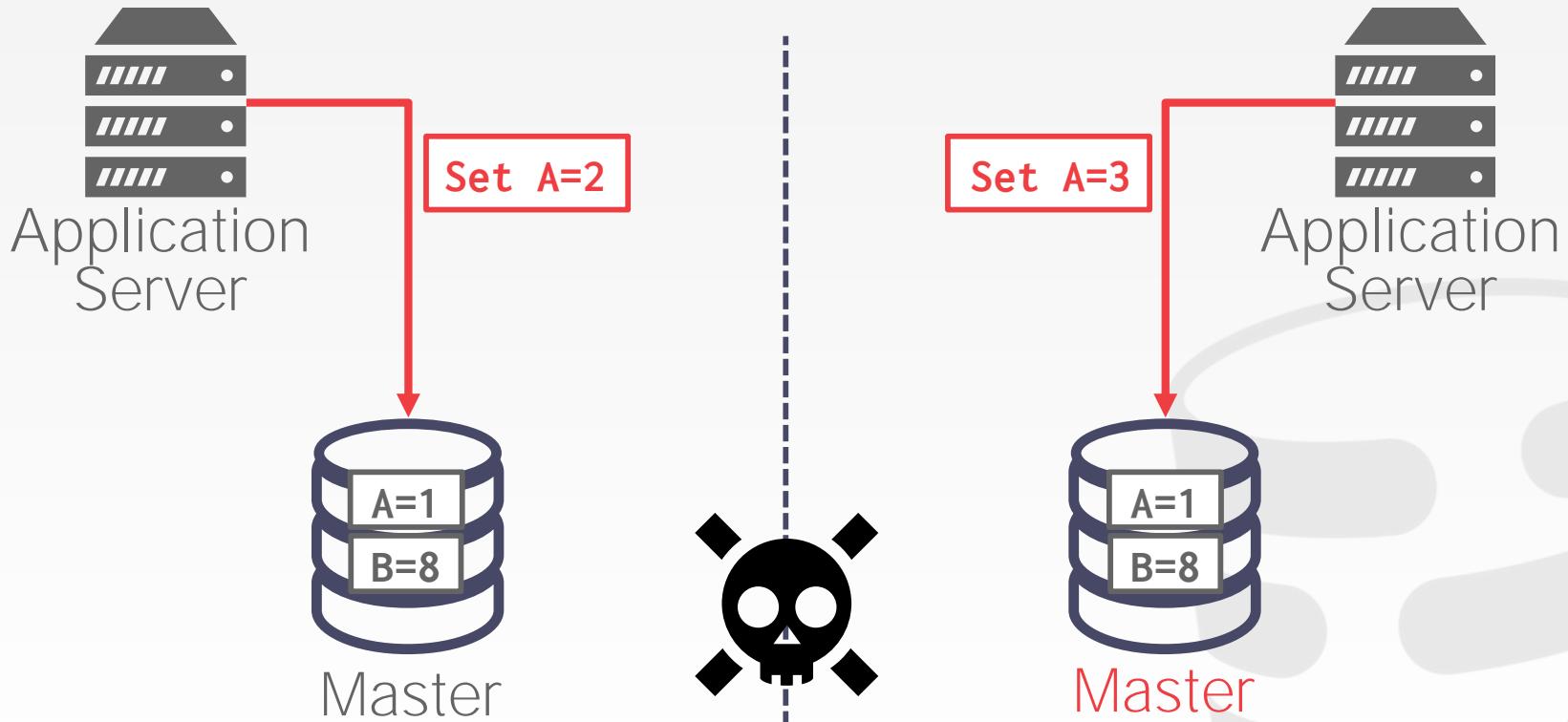
CAP – PARTITION TOLERANCE



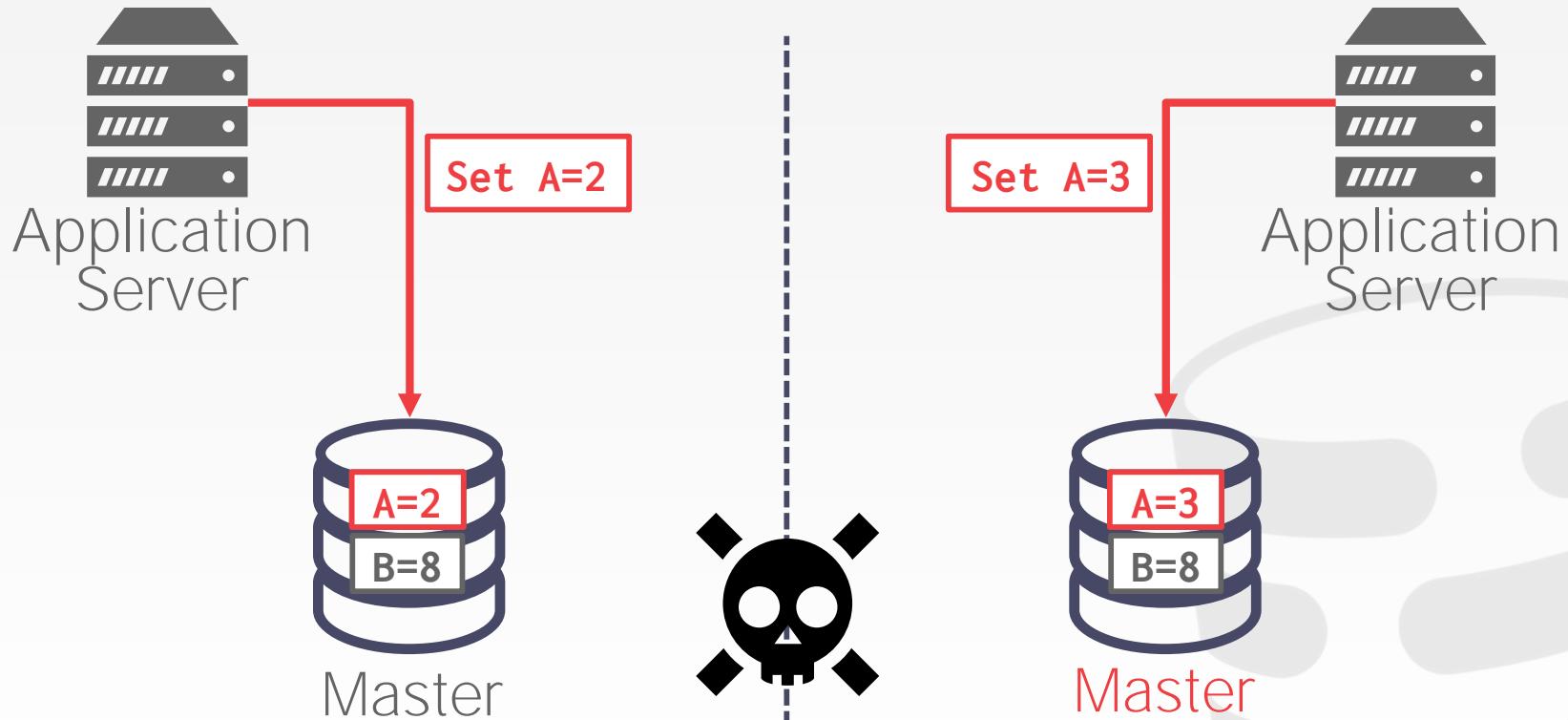
CAP – PARTITION TOLERANCE



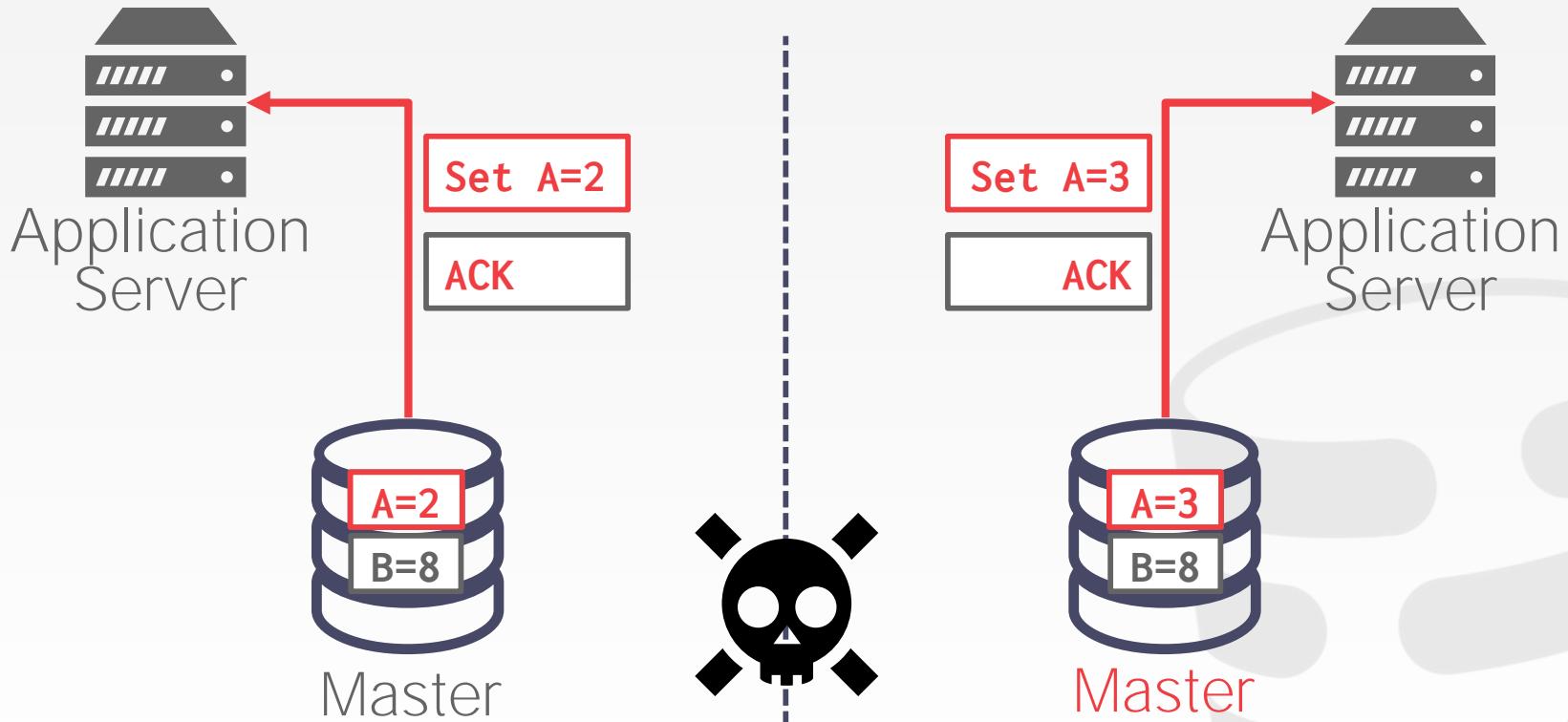
CAP – PARTITION TOLERANCE



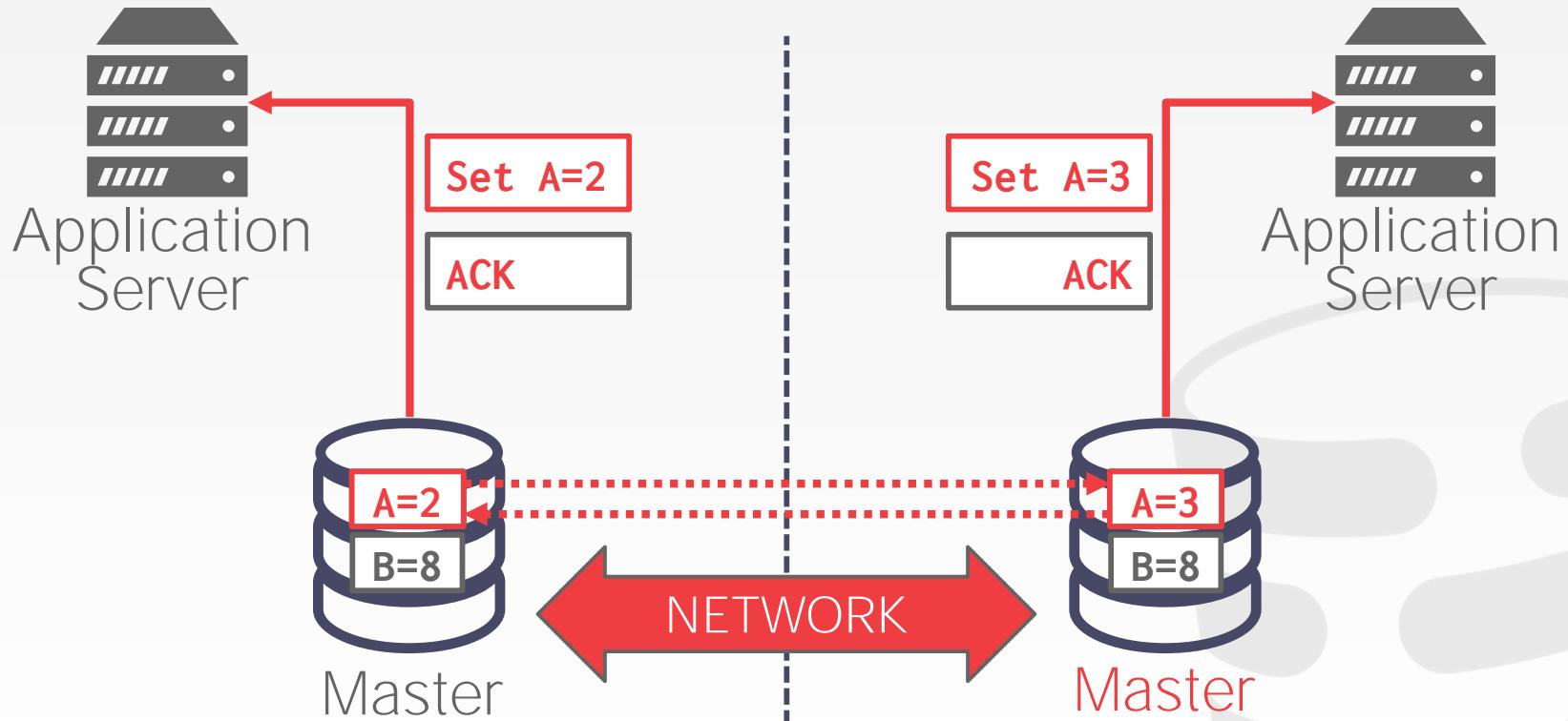
CAP – PARTITION TOLERANCE



CAP – PARTITION TOLERANCE



CAP – PARTITION TOLERANCE



CAP FOR OLTP DBMSS

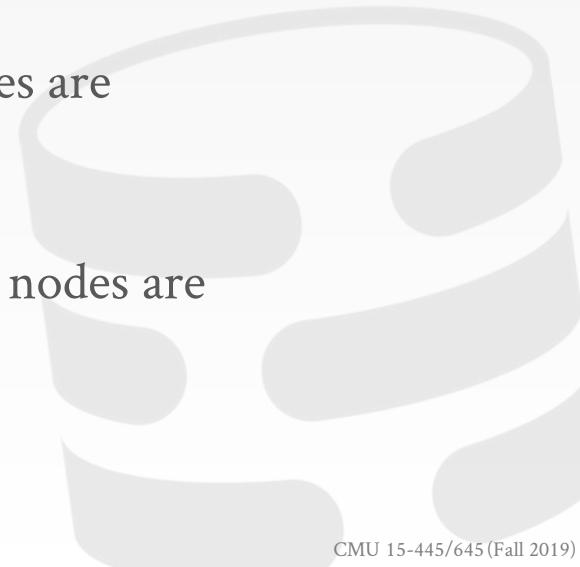
How a DBMS handles failures determines which elements of the CAP theorem they support.

Traditional/NewSQL DBMSS

- Stop allowing updates until a majority of nodes are reconnected.

NoSQL DBMSS

- Provide mechanisms to resolve conflicts after nodes are reconnected.



OBSERVATION

We have assumed that the nodes in our distributed systems are running the same DBMS software.

But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.



FEDERATED DATABASES

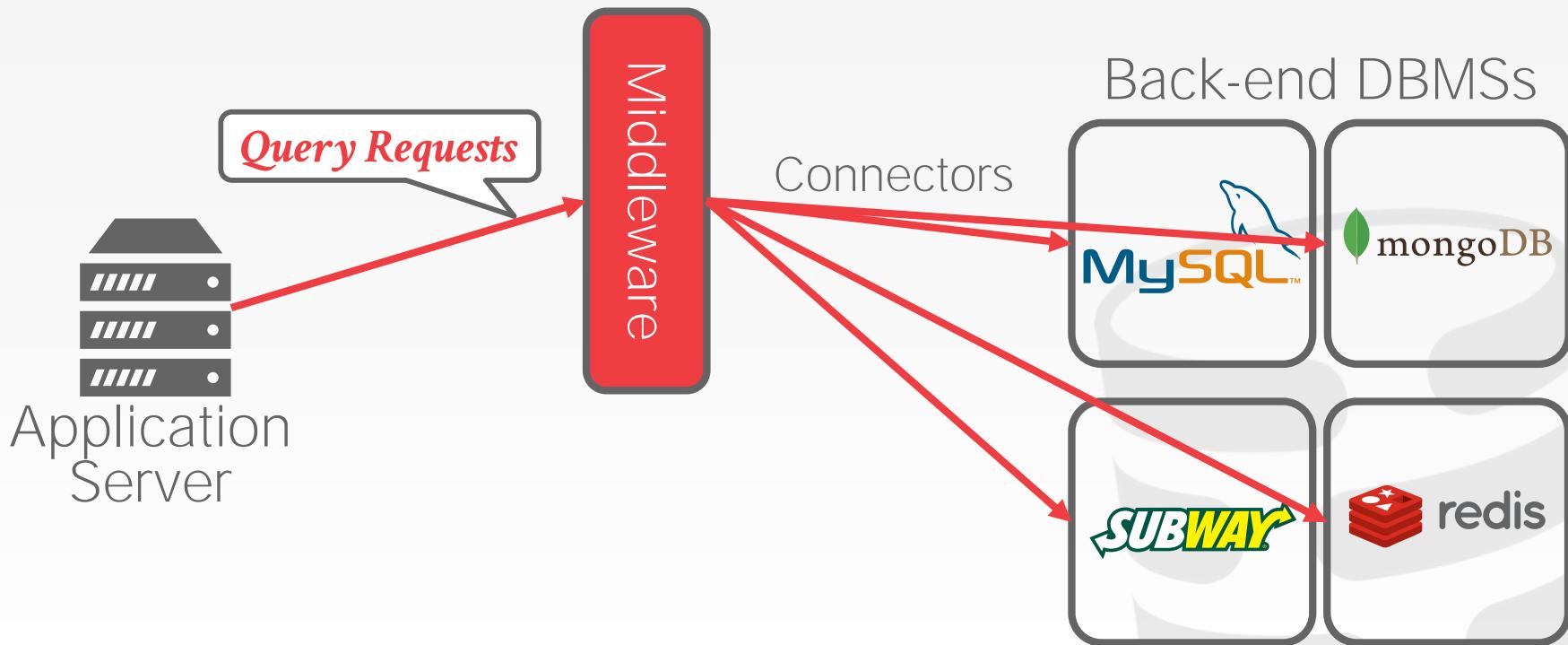
Distributed architecture that connects together multiple DBMSs into a single logical system.
A query can access data at any location.

This is hard and nobody does it well

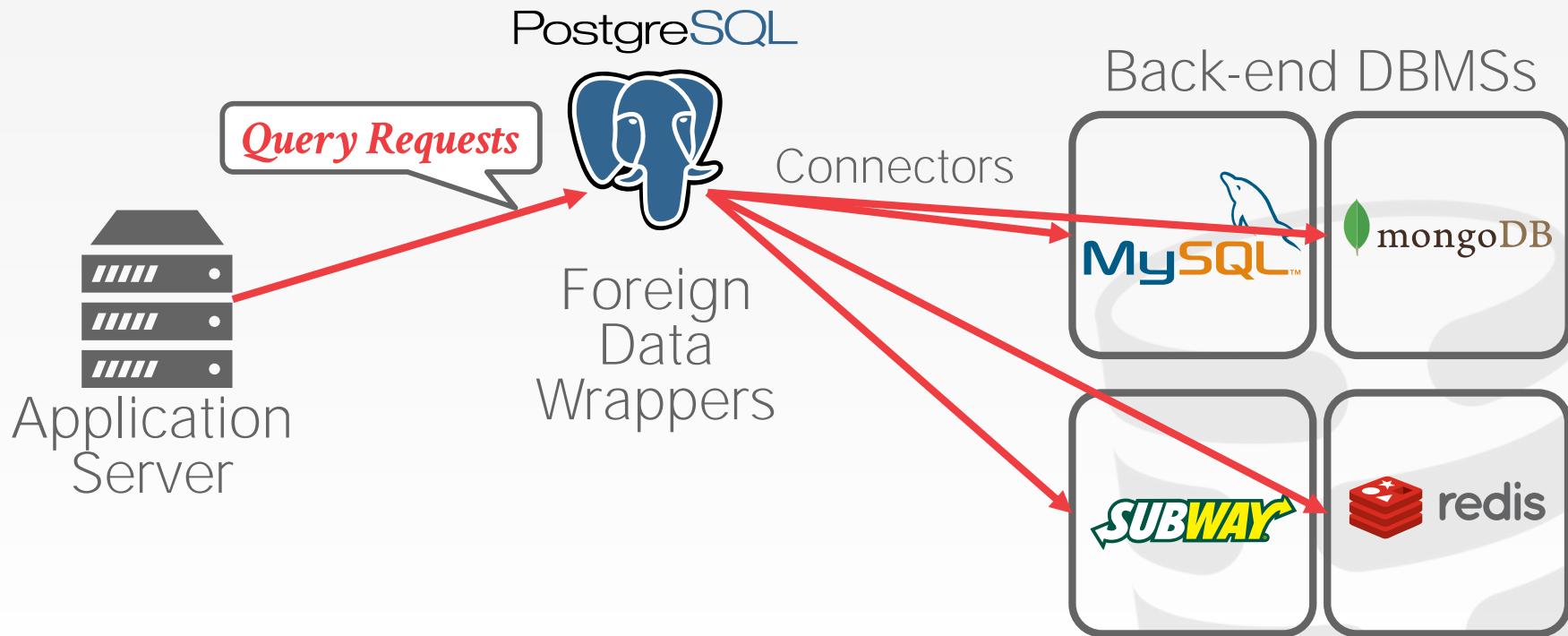
- Different data models, query languages, limitations.
- No easy way to optimize queries
- Lots of data copying (bad).



FEDERATED DATABASE EXAMPLE



FEDERATED DATABASE EXAMPLE



CONCLUSION

We assumed that the nodes in our distributed DBMS are friendly.

Blockchain databases assume that the nodes are adversarial. This means you must use different protocols to commit transactions.



NEXT CLASS

Distributed OLAP Systems



24

Distributed OLAP Databases



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #5: Monday Dec 3rd @ 11:59pm

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 10th @ 11:59pm

Final Exam: Monday Dec 9th @ 5:30pm

Systems Potpourri: Wednesday Dec 4th

- Vote for what system you want me to talk about.
- <https://cmudb.io/f19-systems>

ADMINISTRIVIA

Monday Dec 2nd – Oracle Lecture

→ Shasank Chavan (VP In-Memory Databases)



Monday Dec 2nd – Oracle Systems Talk

→ 4:30pm in GHC 6115

→ Pizza will be served



Tuesday Dec 3rd – Oracle Research Talk

→ Hideaki Kimura (Oracle Beast)

→ 12:00pm in CIC 4th Floor (Panther Hollow Room)

→ Pizza will be served.

LAST CLASS

Atomic Commit Protocols

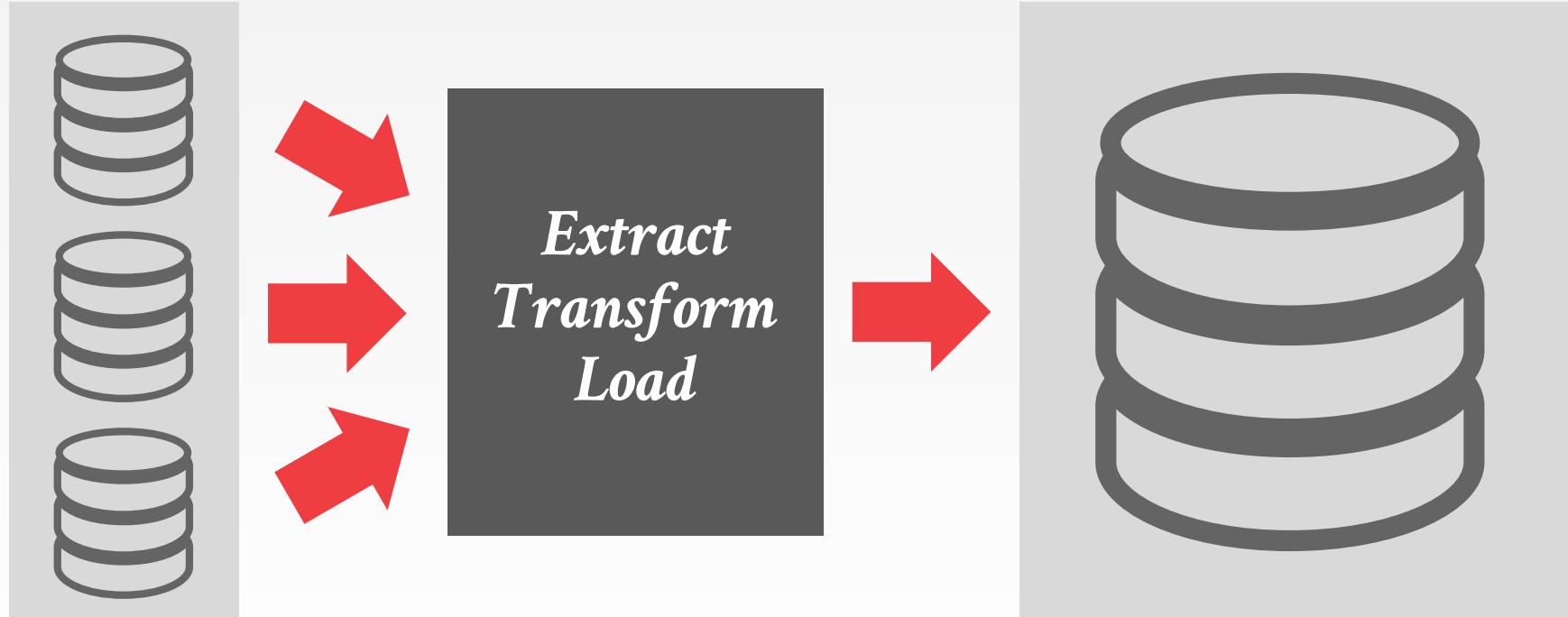
Replication

Consistency Issues (CAP)

Federated Databases



BIFURCATED ENVIRONMENT



OLTP Databases

OLAP Database

DECISION SUPPORT SYSTEMS

Applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analyzing historical data.

Star Schema vs. Snowflake Schema



STAR SCHEMA

PRODUCT_DIM

CATEGORY_NAME
CATEGORY_DESC
PRODUCT_CODE
PRODUCT_NAME
PRODUCT_DESC

SALES_FACT

PRODUCT_FK
TIME_FK
LOCATION_FK
CUSTOMER_FK

PRICE
QUANTITY

LOCATION_DIM

COUNTRY
STATE_CODE
STATE_NAME
ZIP_CODE
CITY

CUSTOMER_DIM

ID
FIRST_NAME
LAST_NAME
EMAIL
ZIP_CODE

TIME_DIM

YEAR
DAY_OF_YEAR
MONTH_NUM
MONTH_NAME
DAY_OF_MONTH

SNOWFLAKE SCHEMA

CAT_LOOKUP

CATEGORY_ID
CATEGORY_NAME
CATEGORY_DESC

STATE_LOOKUP

STATE_ID
STATE_CODE
STATE_NAME

LOCATION_DIM

COUNTRY
STATE_FK
ZIP_CODE
CITY

PRODUCT_DIM

CATEGORY_FK
PRODUCT_CODE
PRODUCT_NAME
PRODUCT_DESC



SALES_FACT

PRODUCT_FK
TIME_FK
LOCATION_FK
CUSTOMER_FK
PRICE
QUANTITY

CUSTOMER_DIM

ID
FIRST_NAME
LAST_NAME
EMAIL
ZIP_CODE

TIME_DIM

YEAR
DAY_OF_YEAR
MONTH_FK
DAY_OF_MONTH

MONTH_LOOKUP

MONTH_NUM
MONTH_NAME
MONTH_SEASON

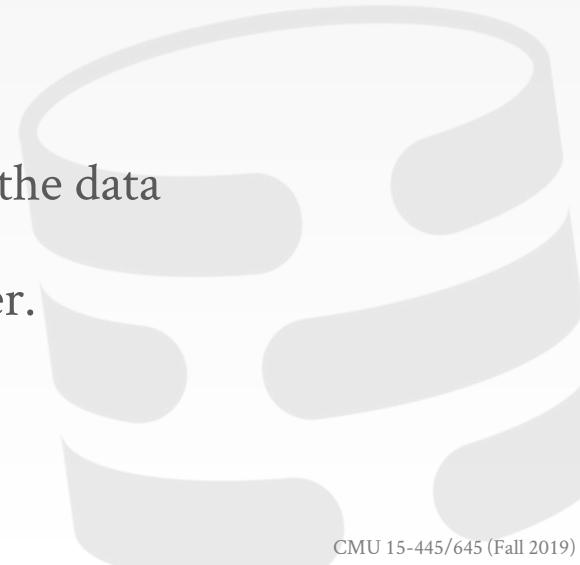
STAR VS. SNOWFLAKE SCHEMA

Issue #1: Normalization

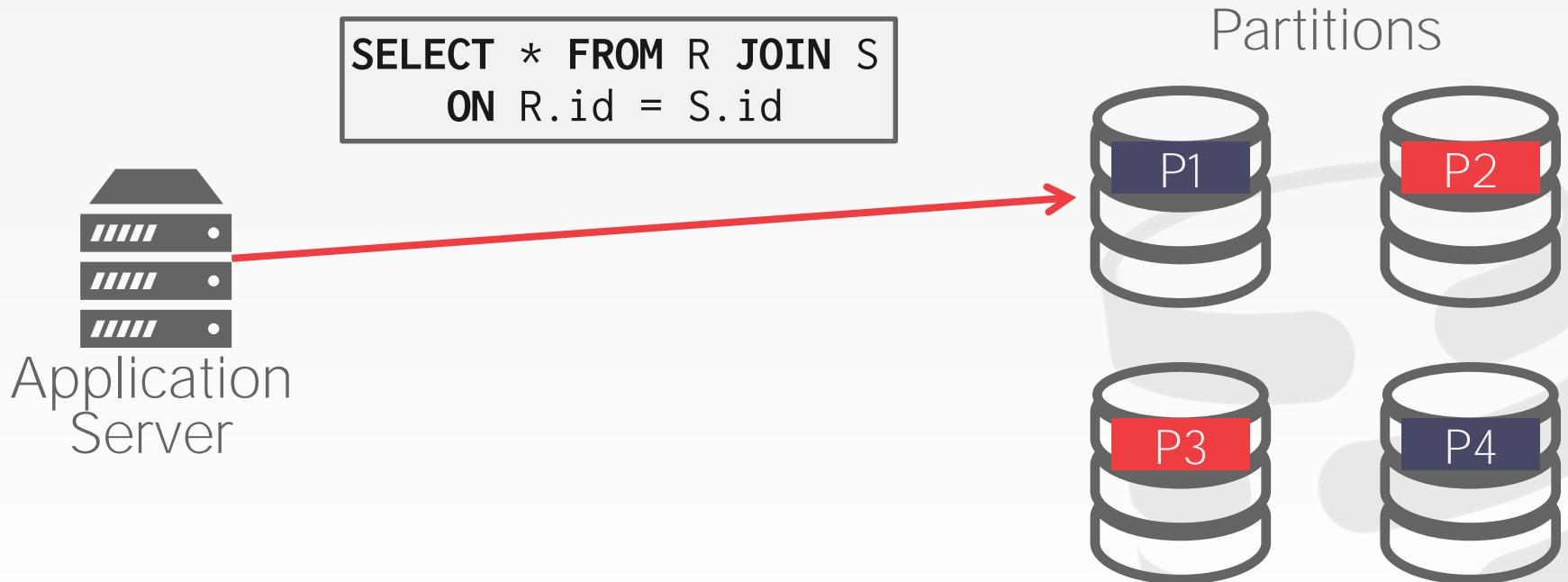
- Snowflake schemas take up less storage space.
- Denormalized data models may incur integrity and consistency violations.

Issue #2: Query Complexity

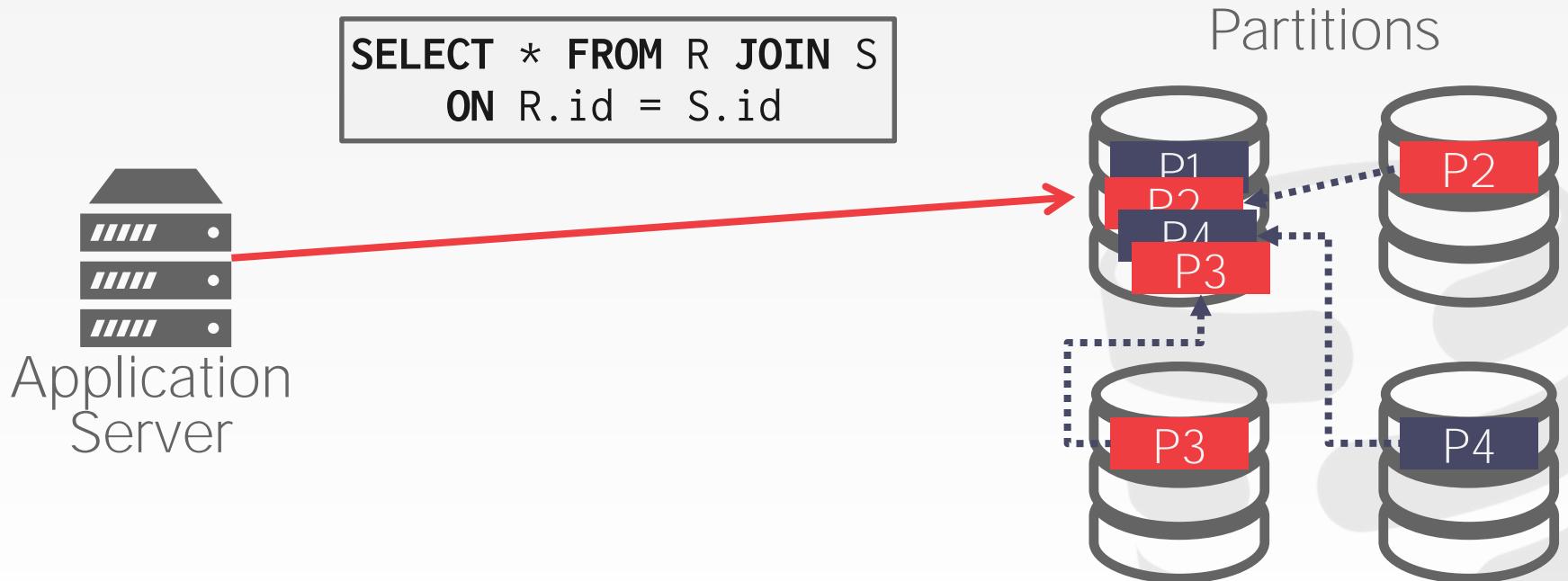
- Snowflake schemas require more joins to get the data needed for a query.
- Queries on star schemas will (usually) be faster.



PROBLEM SETUP



PROBLEM SETUP



TODAY'S AGENDA

Execution Models

Query Planning

Distributed Join Algorithms

Cloud Systems



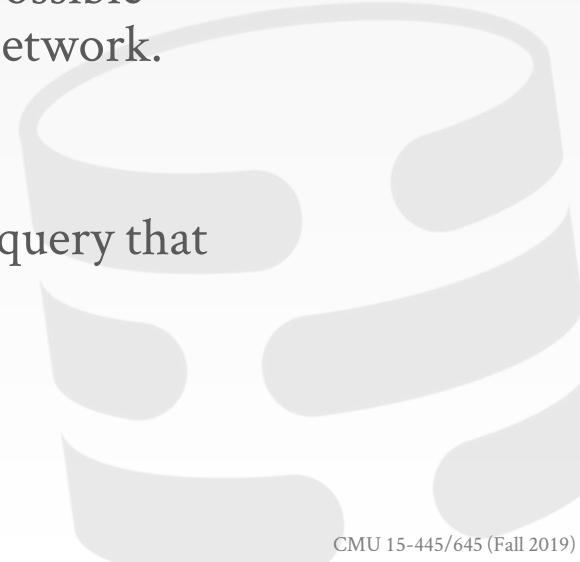
PUSH VS. PULL

Approach #1: Push Query to Data

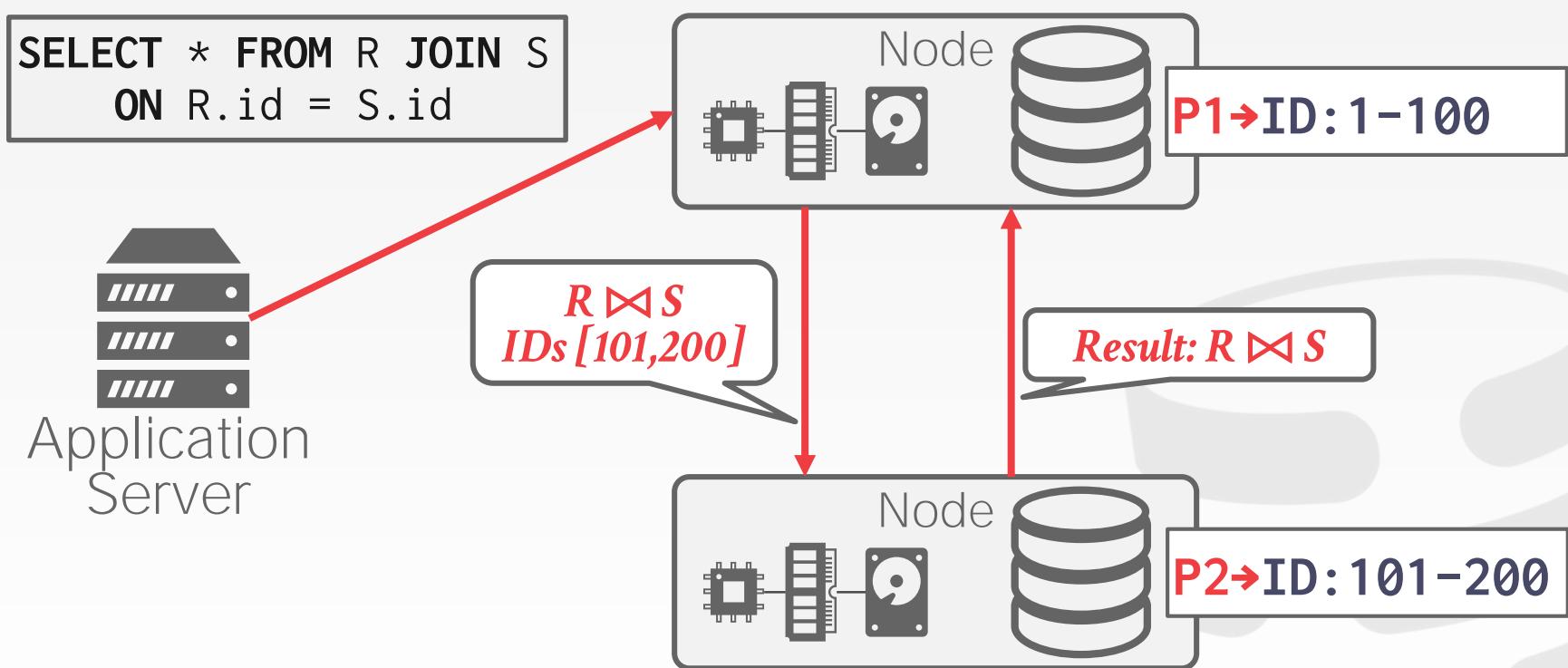
- Send the query (or a portion of it) to the node that contains the data.
- Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query

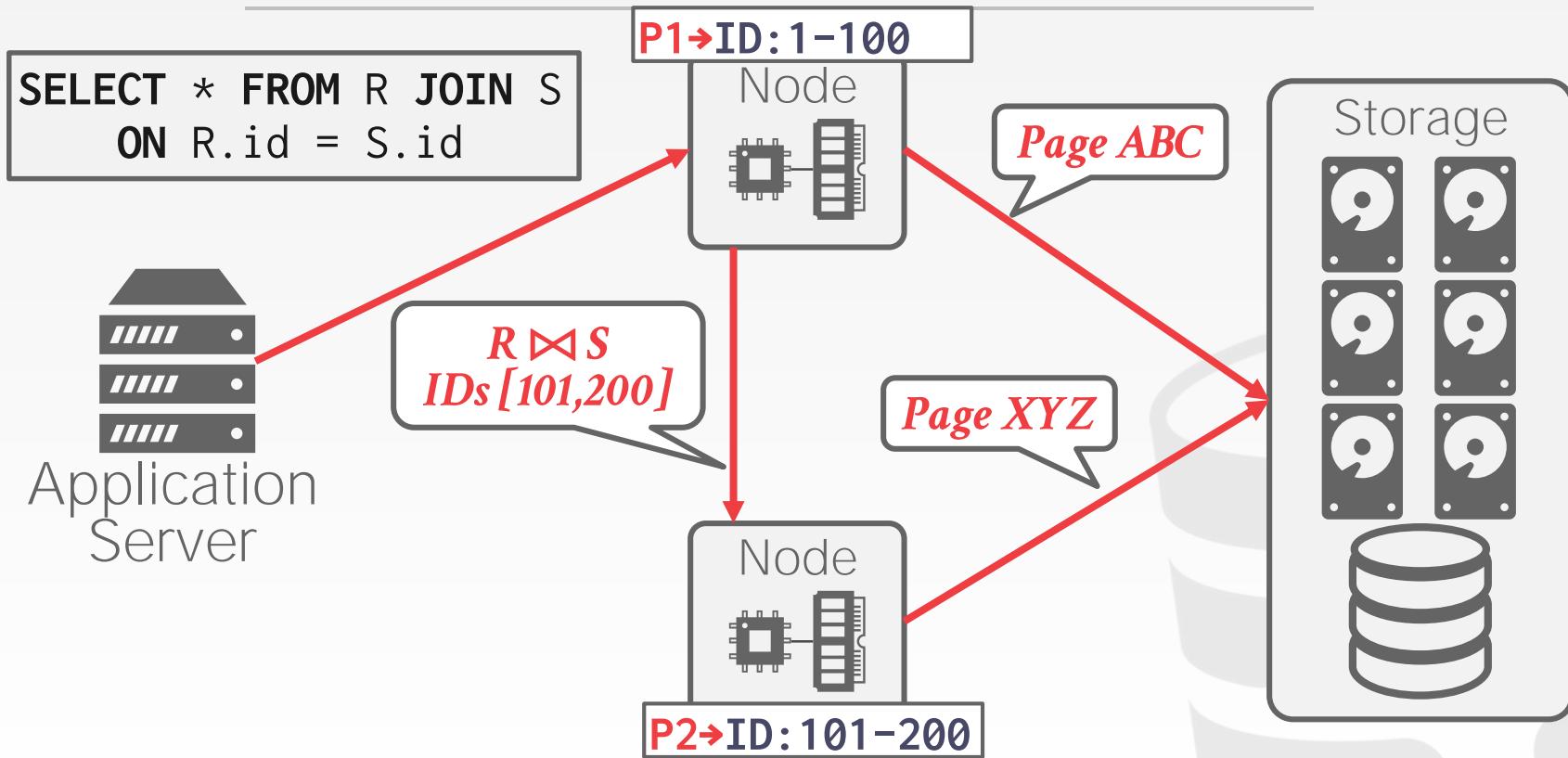
- Bring the data to the node that is executing a query that needs it for processing.



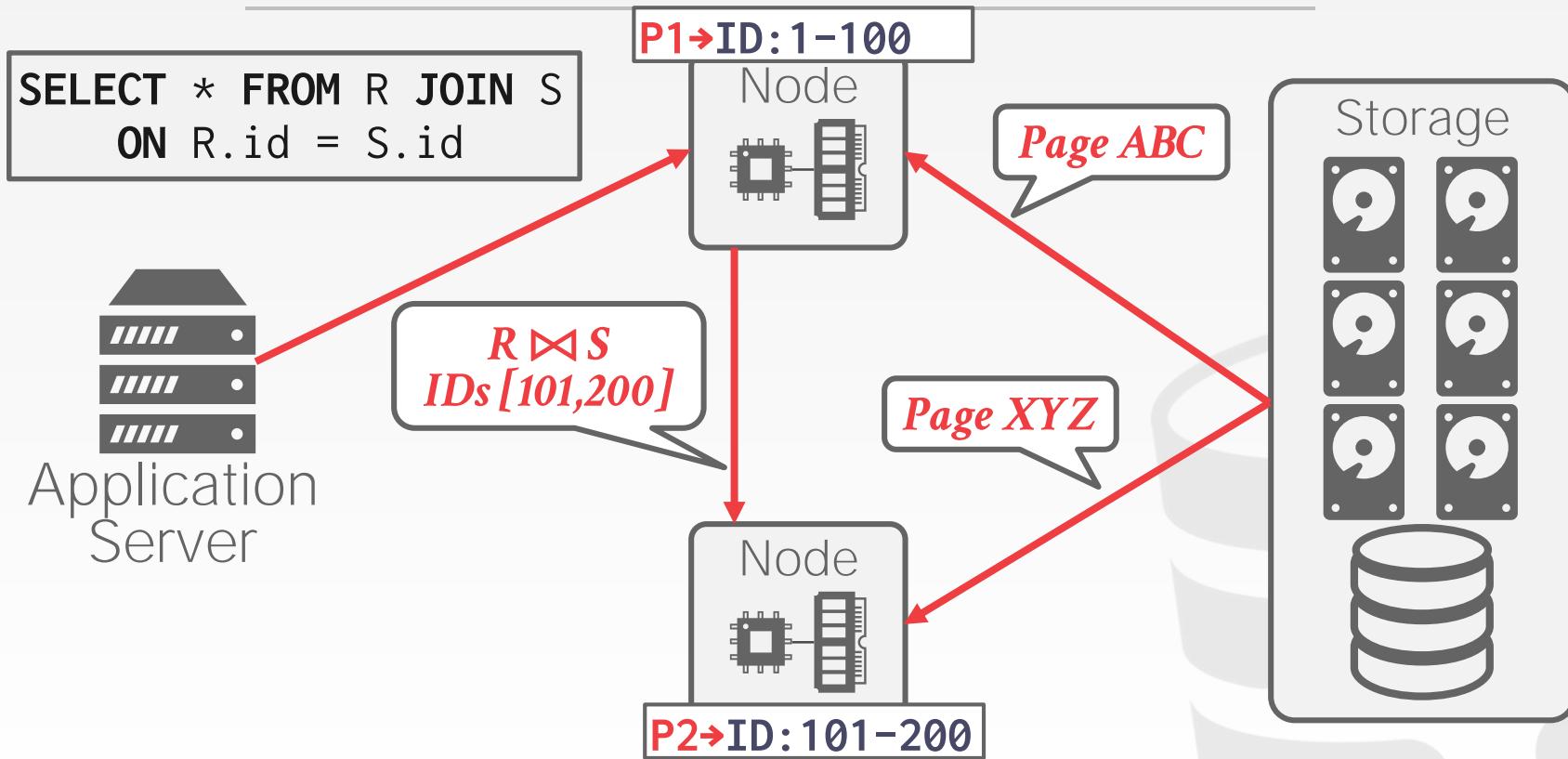
PUSH QUERY TO DATA



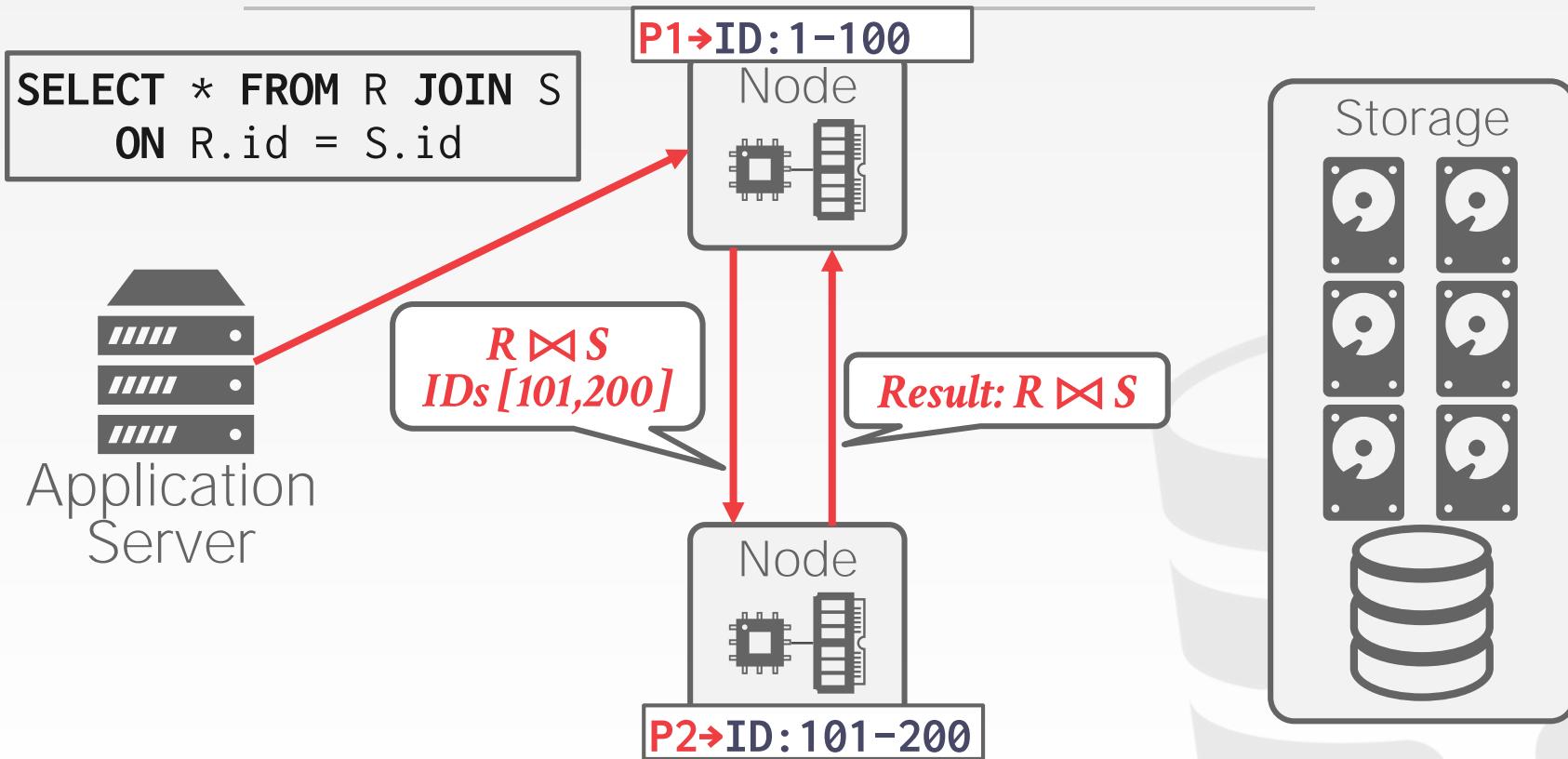
PULL DATA TO QUERY



PULL DATA TO QUERY



PULL DATA TO QUERY

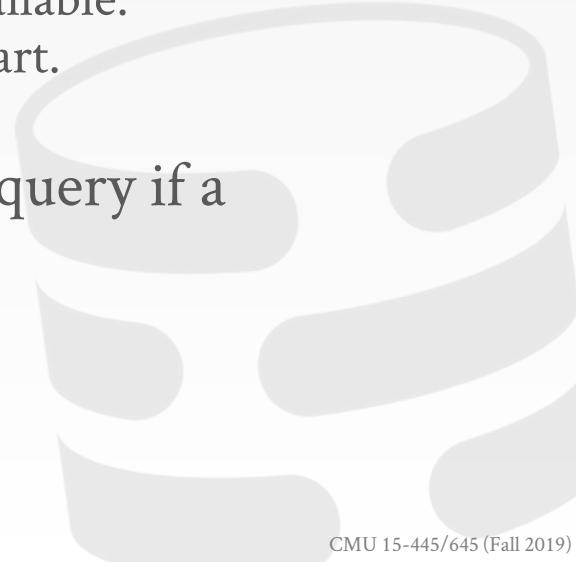


OBSERVATION

The data that a node receives from remote sources are cached in the buffer pool.

- This allows the DBMS to support intermediate results that are large than the amount of memory available.
- Ephemeral pages are not persisted after a restart.

What happens to a long-running OLAP query if a node crashes during execution?

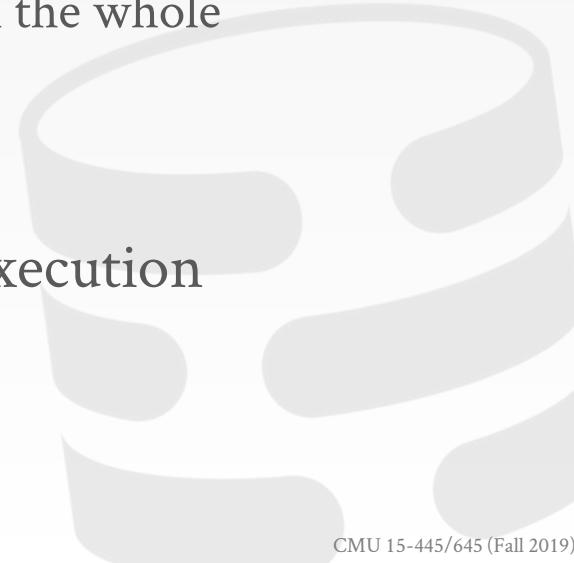


QUERY FAULT TOLERANCE

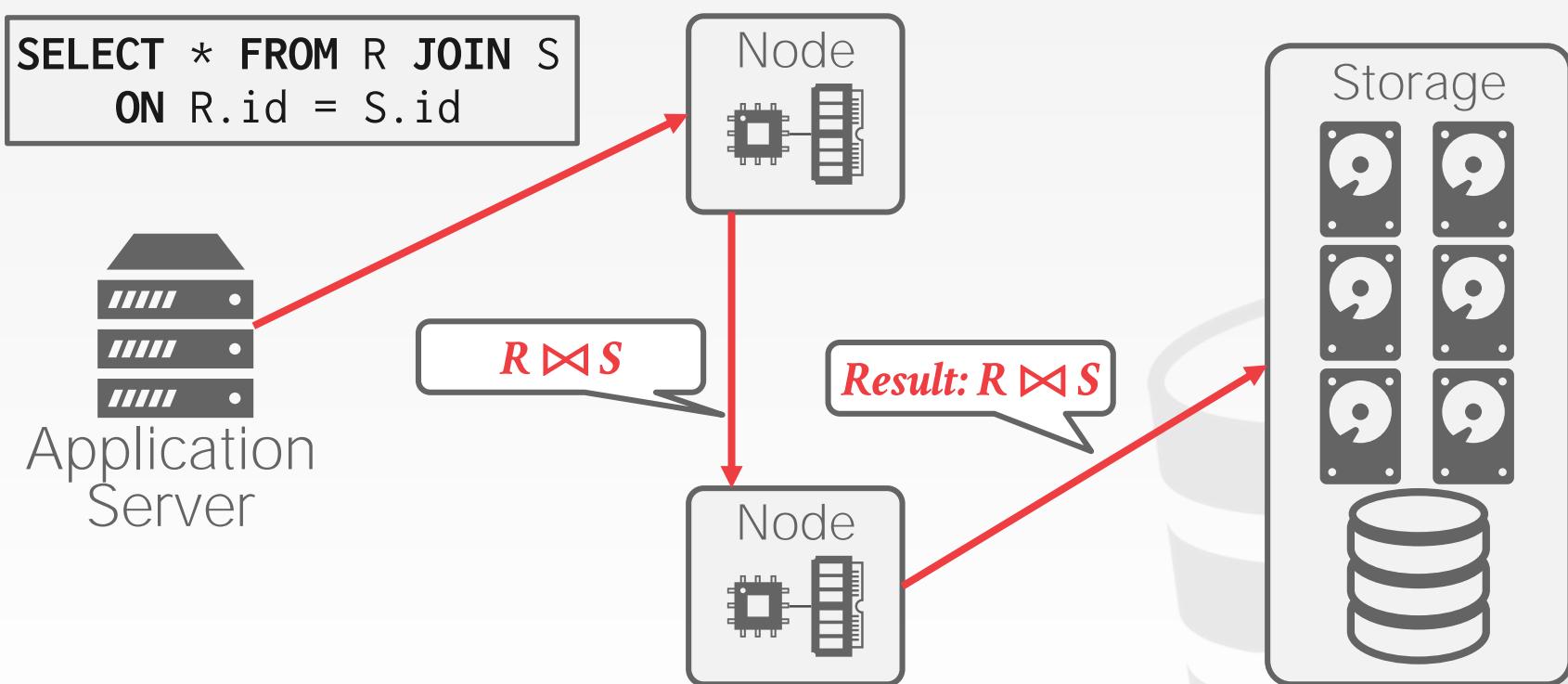
Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution.

→ 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.

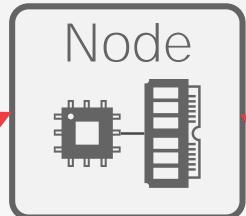


QUERY FAULT TOLERANCE

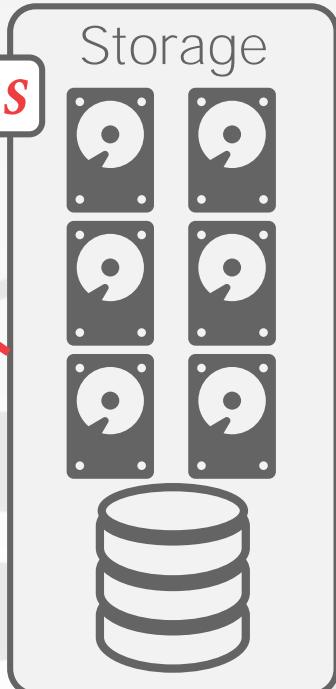


QUERY FAULT TOLERANCE

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



Result: R ⚬ S



QUERY PLANNING

All the optimizations that we talked about before are still applicable in a distributed environment.

- Predicate Pushdown
- Early Projections
- Optimal Join Orderings

Distributed query optimization is even harder because it must consider the location of data in the cluster and data movement costs.



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.

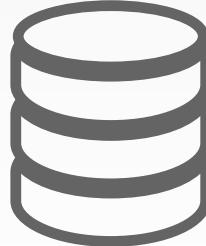
Approach #2: SQL

- Rewrite original query into partition-specific queries.
- Allows for local optimization at each node.
- MemSQL is the only system that I know that does this.

QUERY PLAN FRAGMENTS

```
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
```



Id: 1-100

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



Id: 101-200

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



Id: 201-300

Union the output of each join to produce final result.

N FRAGMENTS

```
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
```



Id: 1-100

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



Id: 101-200

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



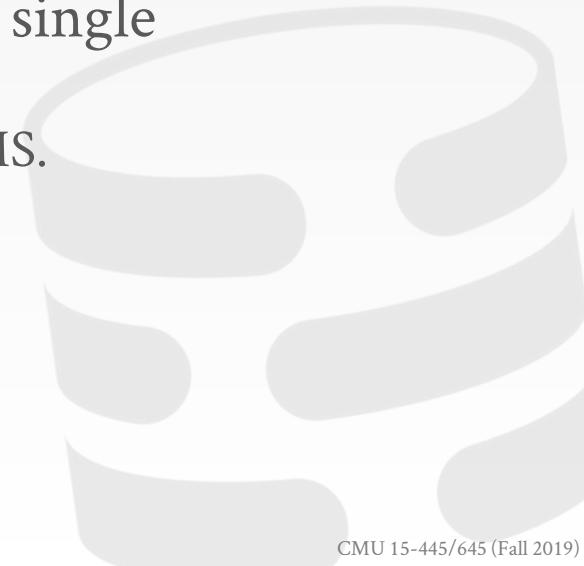
Id: 201-300

OBSERVATION

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.

- You lose 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.

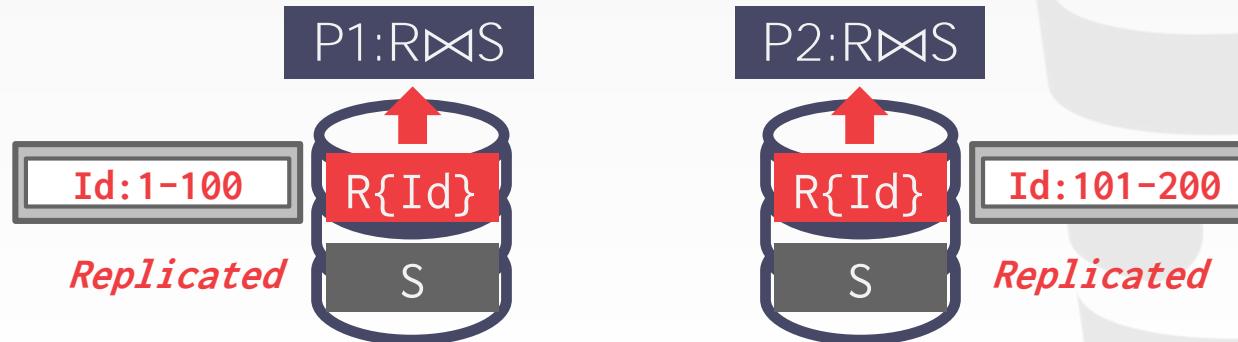
Once there, it then executes the same join algorithms that we discussed earlier in the semester.



SCENARIO #1

One table is replicated at every node.
Each node joins its local data and then
sends their results to a coordinating
node.

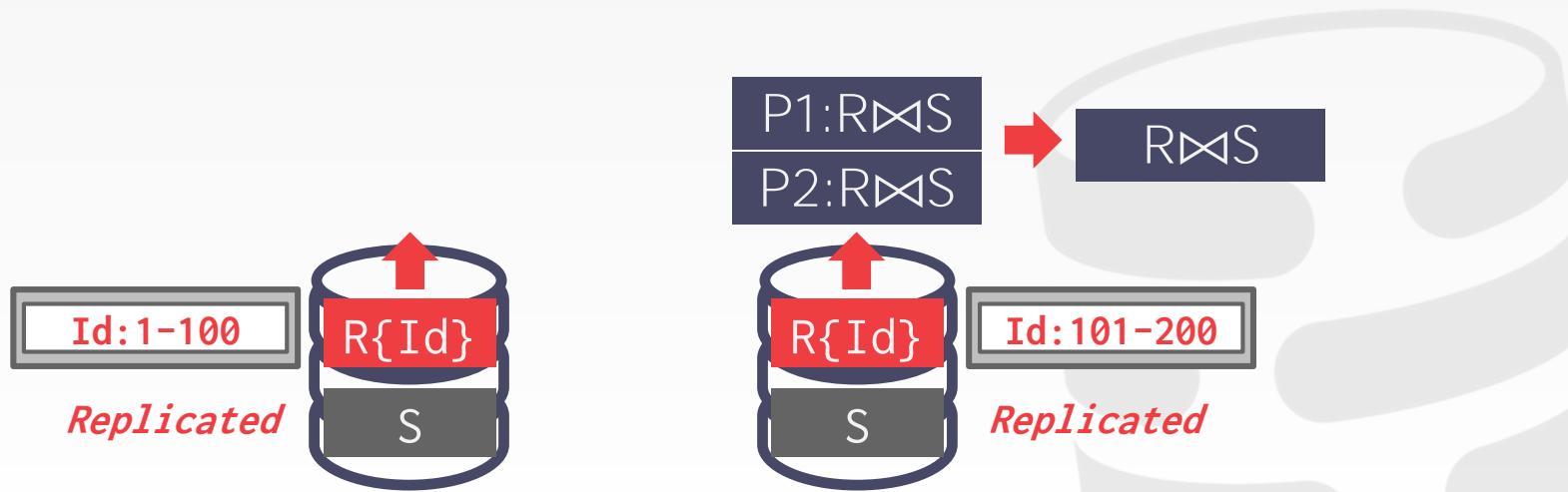
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #1

One table is replicated at every node.
Each node joins its local data and then
sends their results to a coordinating
node.

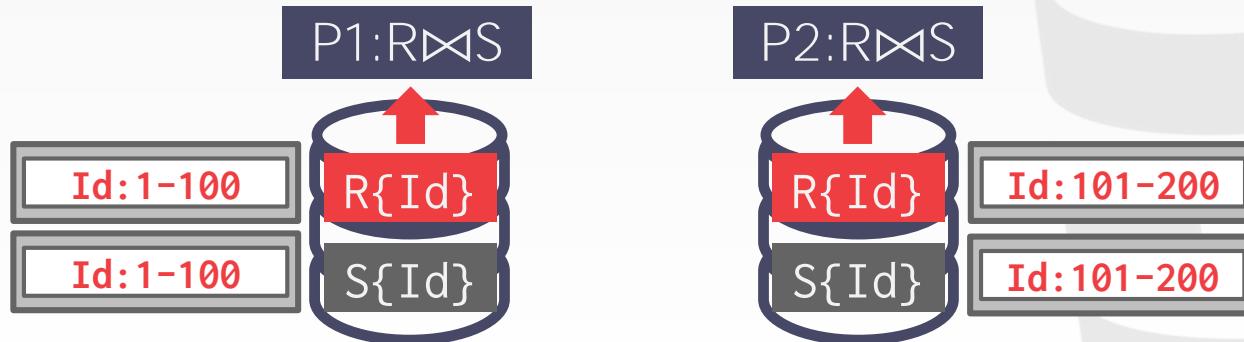
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a node for coalescing.

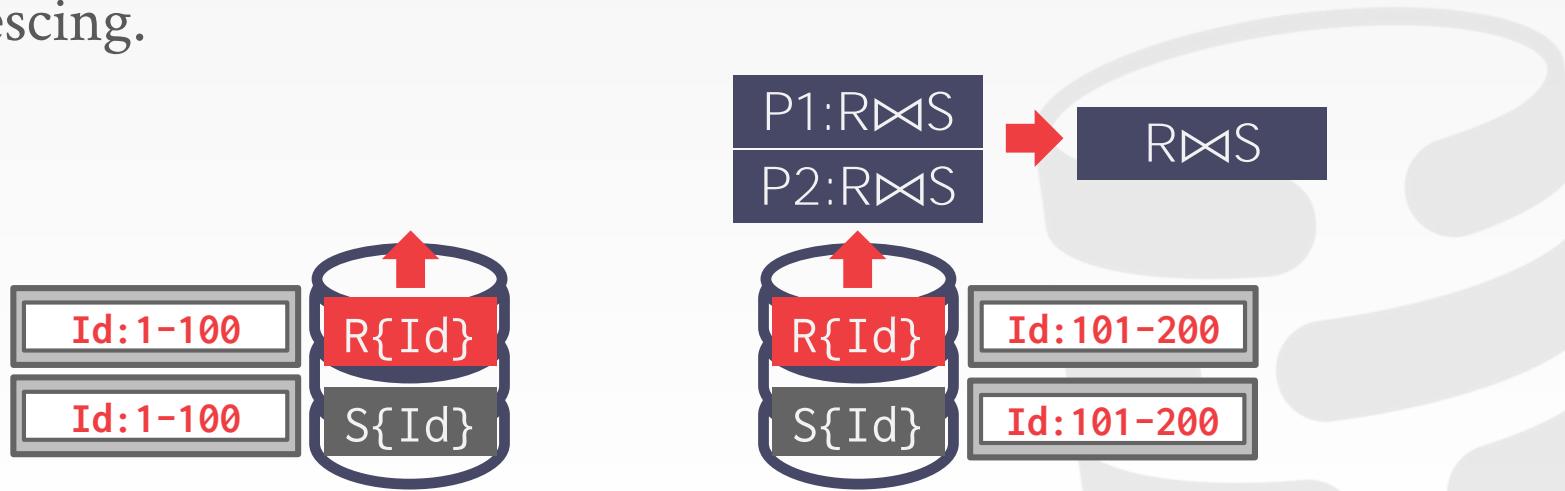
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a 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.

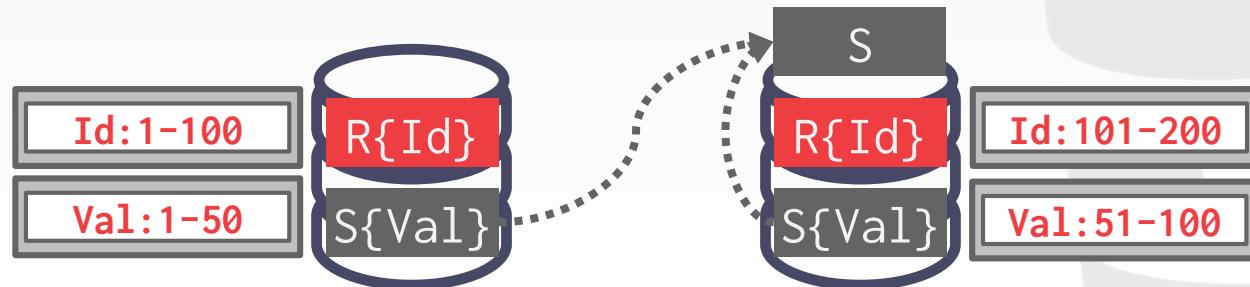
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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.

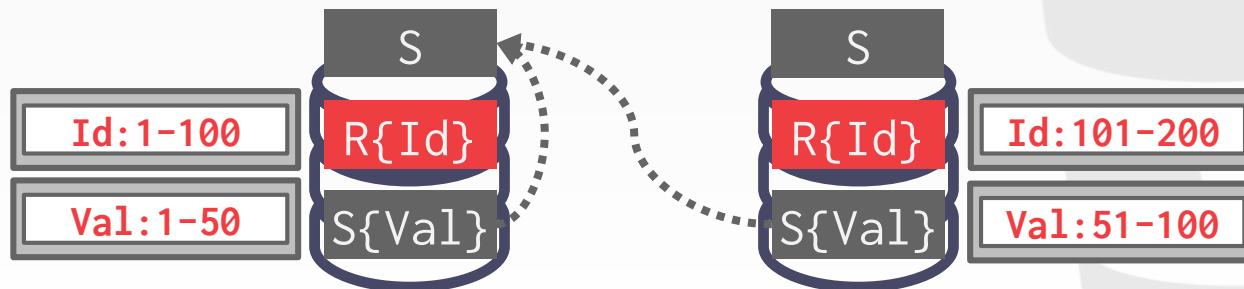
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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.

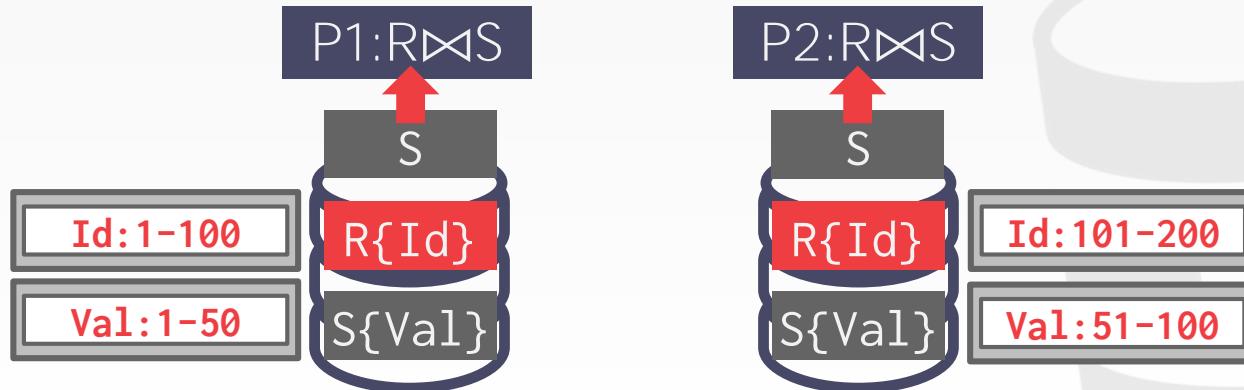
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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.

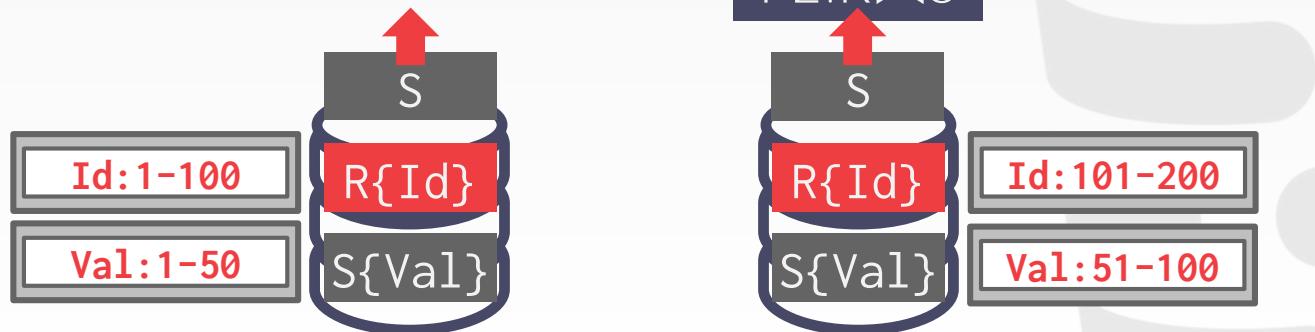
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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.

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



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

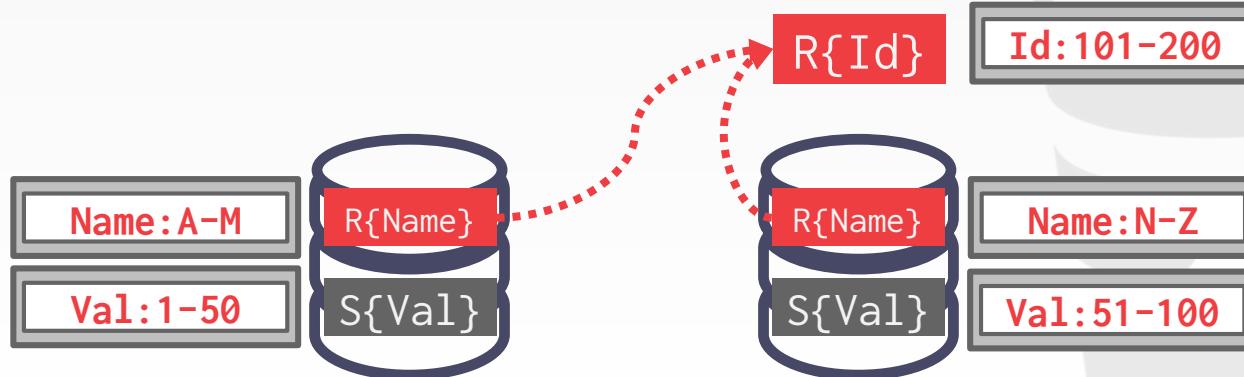
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

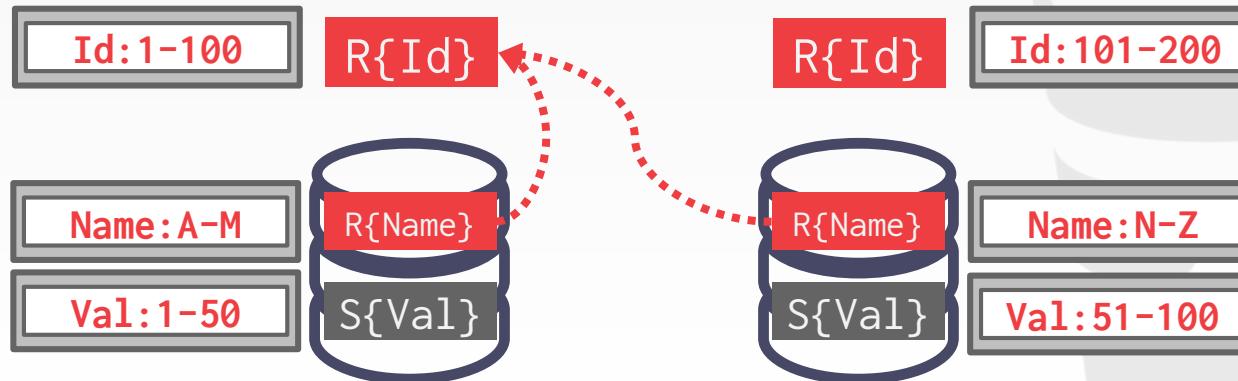
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

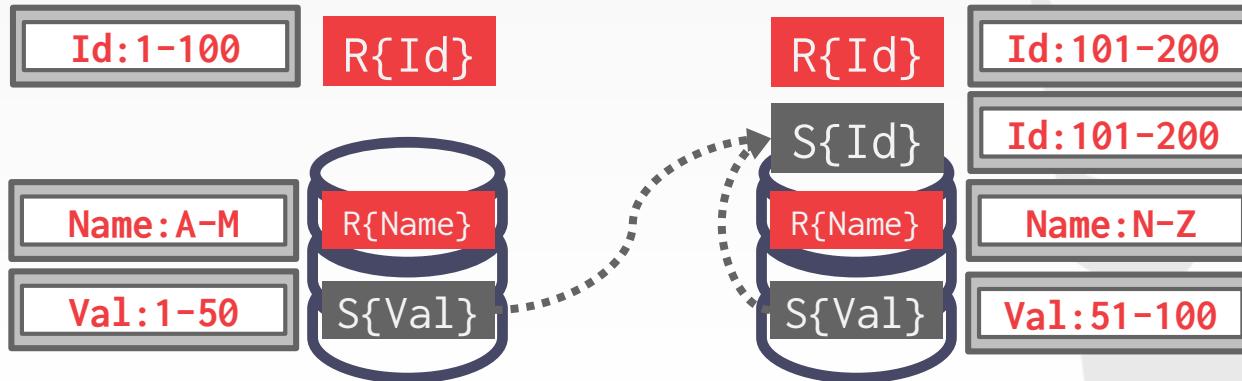
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

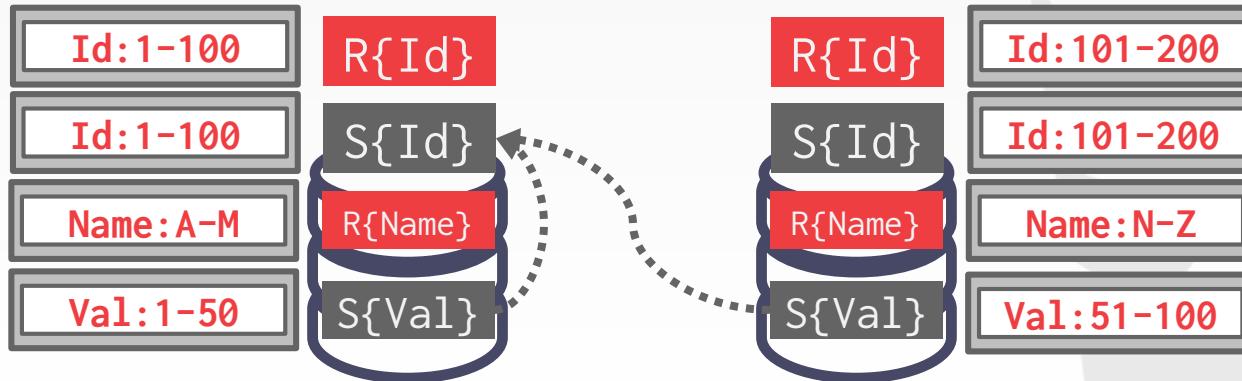
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

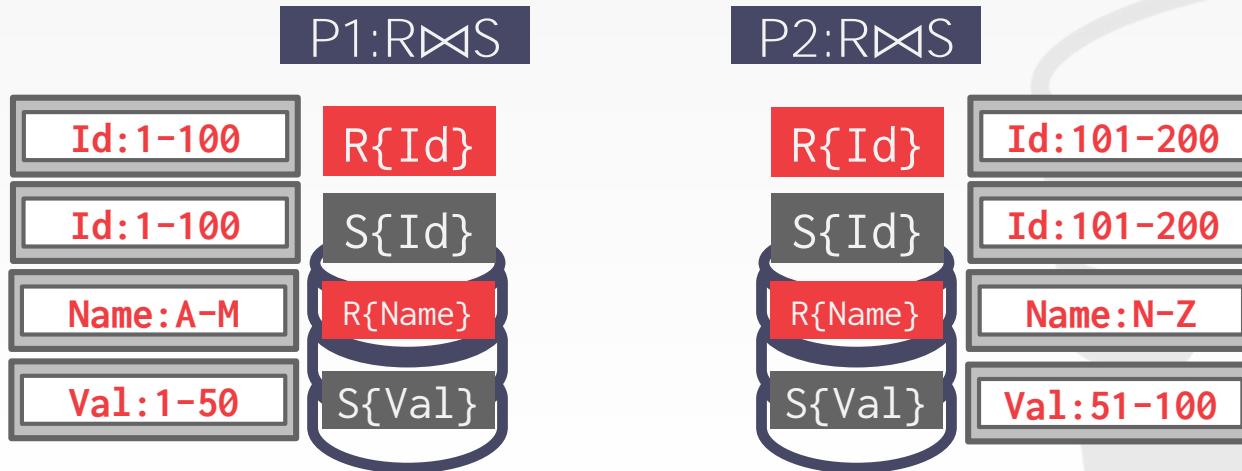
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

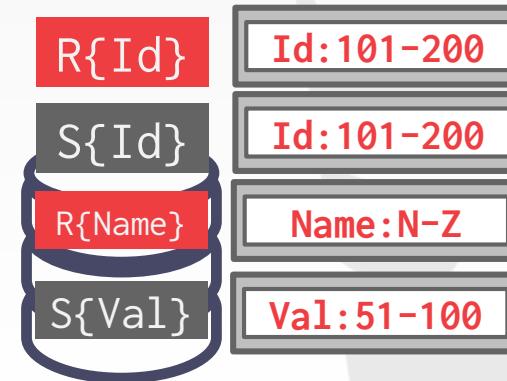
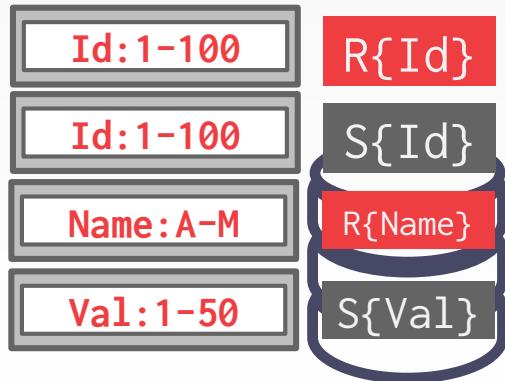
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

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



SEMI-JOIN

Join operator where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id FROM R  
LEFT OUTER JOIN S  
ON R.id = S.id  
WHERE R.id IS NOT NULL
```



SEMI-JOIN

Join operator where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id FROM R  
LEFT OUTER JOIN S  
ON R.id = S.id  
WHERE R.id IS NOT NULL
```



SEMI-JOIN

Join operator where the result only contains columns from the left table.
 Distributed DBMSs use semi-join to minimize the amount of data sent during joins.
 → This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id FROM R
LEFT OUTER JOIN S
ON R.id = S.id
WHERE R.id IS NOT NULL
```



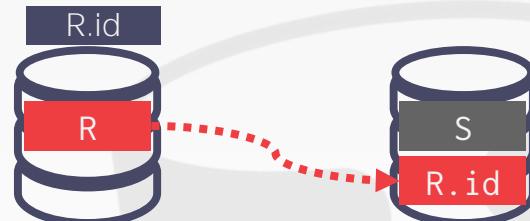
```
SELECT R.id FROM R
WHERE EXISTS (
  SELECT 1 FROM S
  WHERE R.id = S.id)
```

SEMI-JOIN

Join operator where the result only contains columns from the left table.
 Distributed DBMSs use semi-join to minimize the amount of data sent during joins.
 → This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id FROM R
LEFT OUTER JOIN S
ON R.id = S.id
WHERE R.id IS NOT NULL
```



```
SELECT R.id FROM R
WHERE EXISTS (
    SELECT 1 FROM S
    WHERE R.id = S.id)
```

RELATIONAL ALGEBRA: SEMI-JOIN

Like a natural join except that the attributes on the right table that are not used to compute the join are restricted.

Syntax: $(R \ltimes S)$

$R(a_{id}, b_{id}, xxx)$ $S(a_{id}, b_{id}, yyy)$

a_id	b_id	xxx
a1	101	X1
a2	102	X2
a3	103	X3

a_id	b_id	yyy
a3	103	Y1
a4	104	Y2
a5	105	Y3

$(R \ltimes S)$

a_id	b_id	xxx
a3	103	X3

CLOUD SYSTEMS

Vendors provide *database-as-a-service* (DBaaS) offerings that are managed DBMS environments.

Newer systems are starting to blur the lines between shared-nothing and shared-disk.

→ Example: You can do simple filtering on Amazon S3 before copying data to compute nodes.

CLOUD SYSTEMS

Approach #1: Managed DBMSs

- No significant modification to the DBMS to be "aware" that it is running in a cloud environment.
- Examples: Most vendors

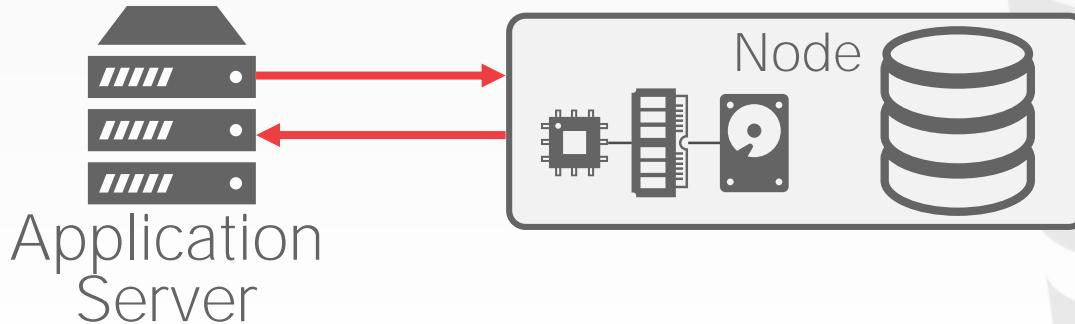
Approach #2: Cloud-Native DBMS

- The system is designed explicitly to run in a cloud environment.
- Usually based on a shared-disk architecture.
- Examples: Snowflake, Google BigQuery, Amazon Redshift, Microsoft SQL Azure



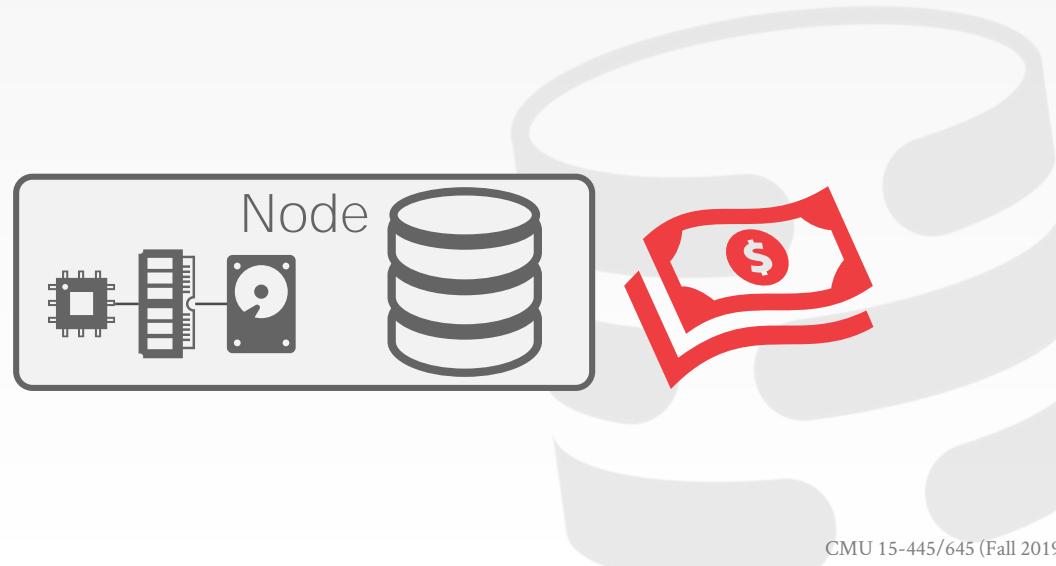
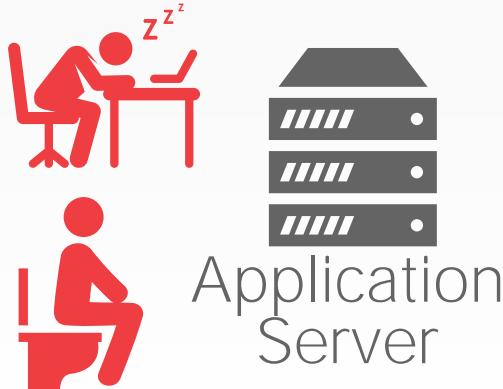
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



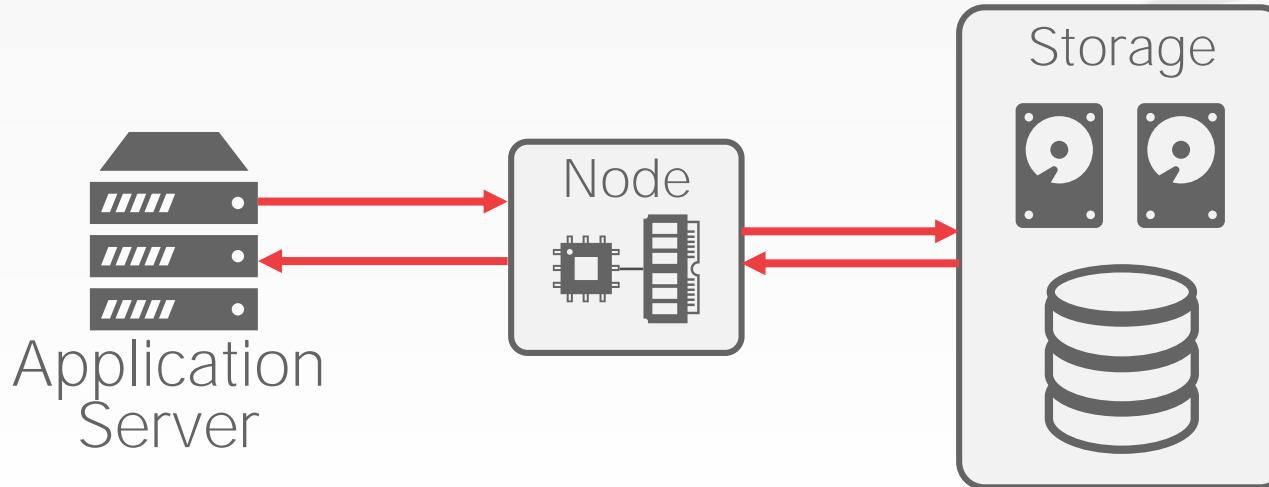
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



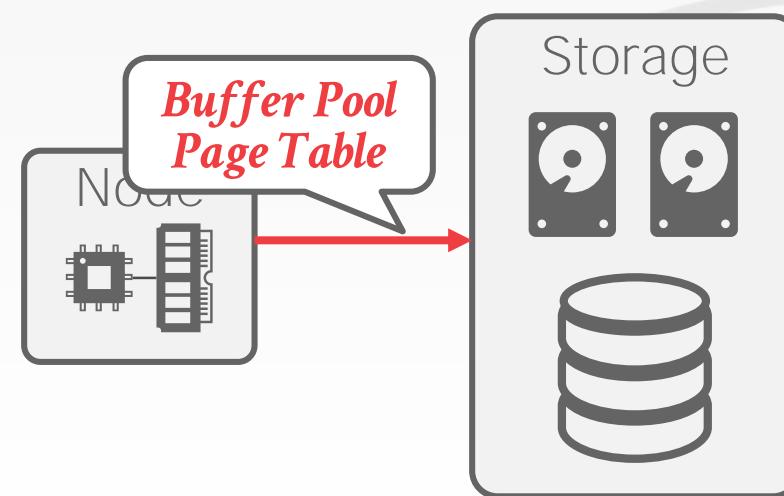
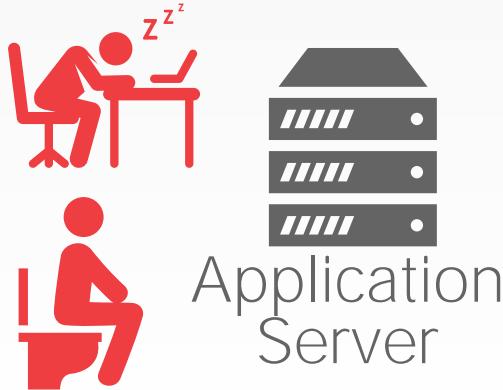
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



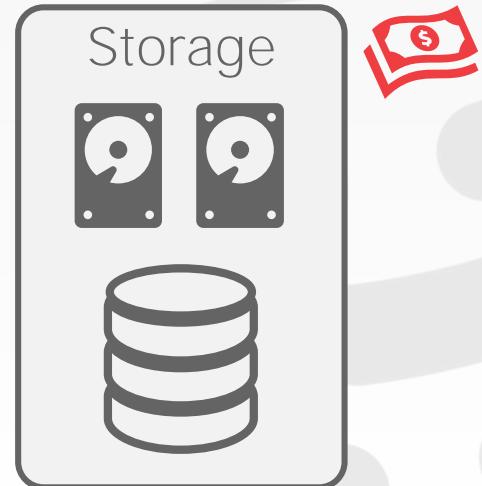
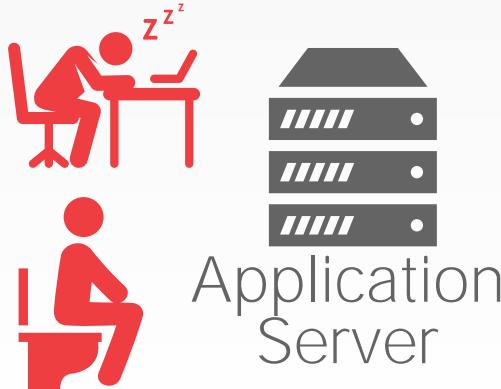
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



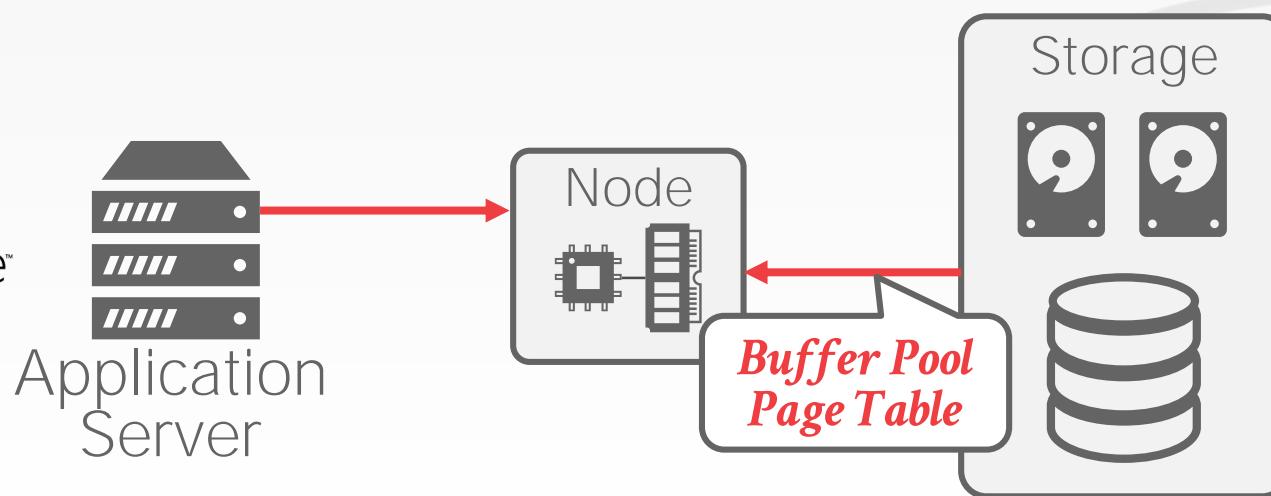
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.

amazon

FAUNA

Microsoft®
SQL Azure™



DISAGGREGATED COMPONENTS

System Catalogs

- [HCatalog](#), [Google Data Catalog](#), [Amazon Glue Data Catalog](#)

Node Management

- [Kubernetes](#), [Apache YARN](#), Cloud Vendor Tools

Query Optimizers

- [Greenplum Orca](#), [Apache Calcite](#)



UNIVERSAL FORMATS

Most DBMSs use a proprietary on-disk binary file format for their databases.

→ Think of the BusTub page types...

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.

UNIVERSAL 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.

CONCLUSION

More money, more data, more problems...



Cloud OLAP Vendors:



**amazon
REDSHIFT**

ORACLE



snowflake



Microsoft[®]
SQL Azure

On-Premise OLAP Systems:



ClickHouse

**splice
MACHINE**



presto



Greenplum

VERTICA

**ORACLE
EXADATA**

TERADATA

NEXT CLASS

Oracle Guest Speaker





ORACLE

Top-5 Innovations of Oracle's Database In-Memory

CMU, 2019

Shasank Chavan

Vice President, In-Memory Database Technologies



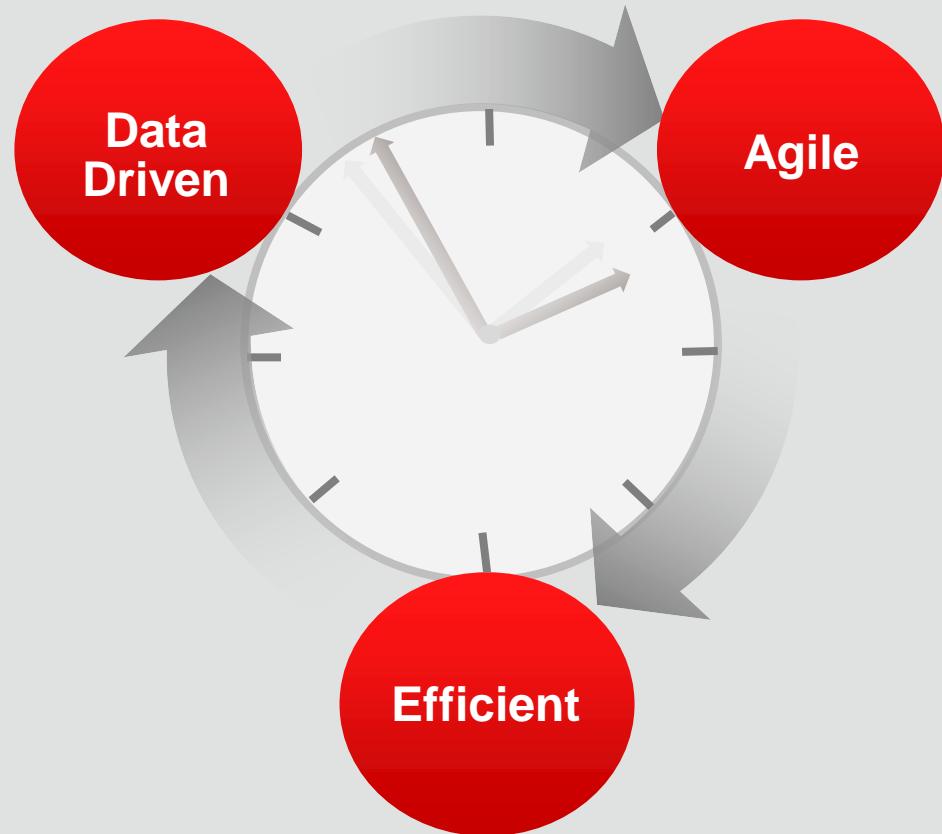
Safe Harbor

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle's future plans, expectations, beliefs, intentions and prospects are "forward-looking statements" and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle's Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading "Risk Factors." These filings are available on the SEC's website or on Oracle's website at <http://www.oracle.com/investor>. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.



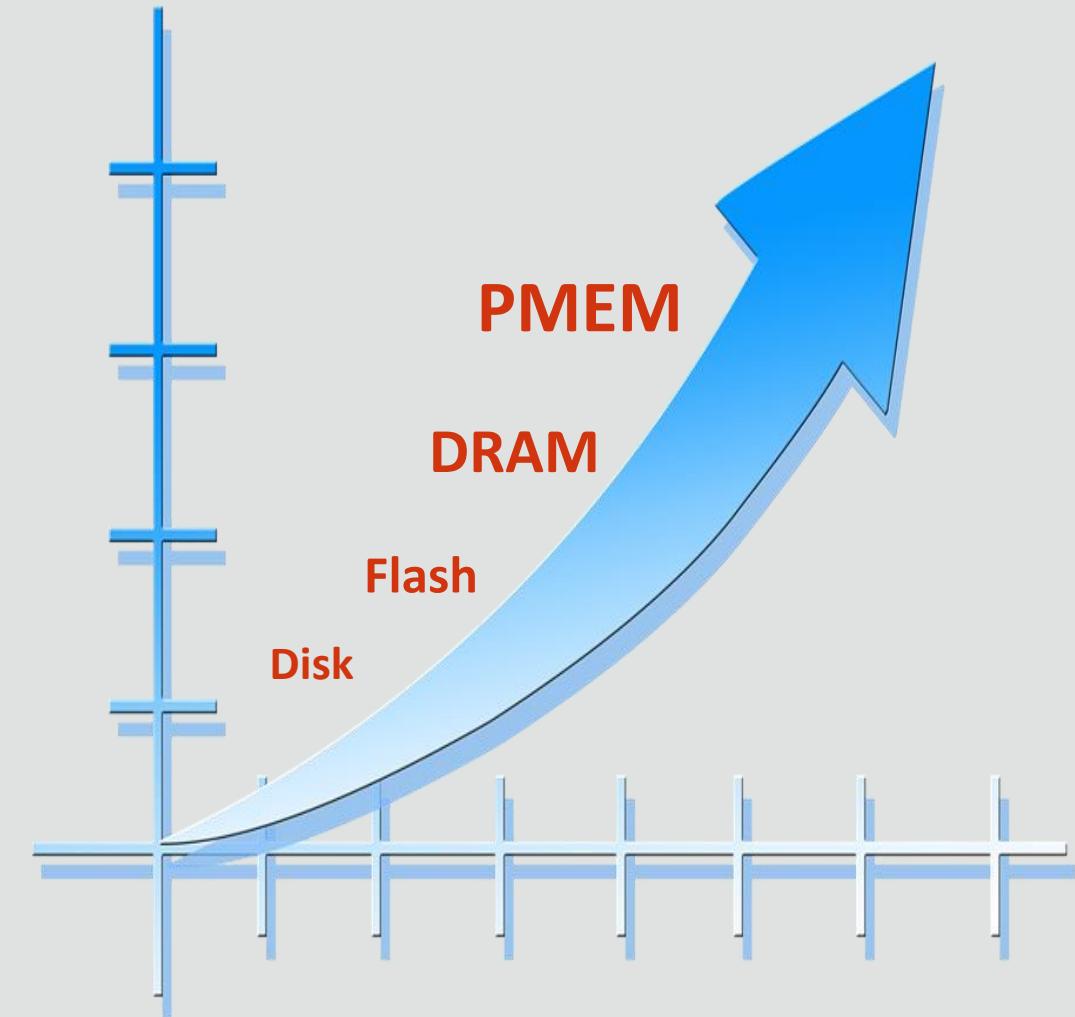
In-Memory Now | Real-Time Enterprises



- **Insurance companies** improve portfolios and reduce cost with real-time analytics for pricing
- **Retailers** use location-based analytics to automate sending personalized mobile coupons to customers
- **Manufacturing Processes** use real-time analytics to monitor production quality and adjust assembly parameters
- **Financial Services** perform risk/fraud analysis across channels in real-time, not after the event occurs
- **Telecom and Broadband** vendors use real-time congestion metrics to optimize their networks

In-Memory Now | Hardware Trends

- **Larger, Cheaper Memory (DRAM, PMEM)**
- **Larger CPU Caches** (e.g. 32MB Shared L3 Cache)
- **Larger Multi-Core Processors** (24 cores w/ Intel)
- **Larger SIMD Vector Processing Units** (e.g. AVX-512)
- **Faster Networks** (100Gb/s RoCE vs 40Gb/s Infiniband)
- **NUMA Architectures** (Local Memory vs Remote)
- **Persistent Memory** (Availability, Capacity, Speed)



In-Memory Now | Technology Across All Tiers

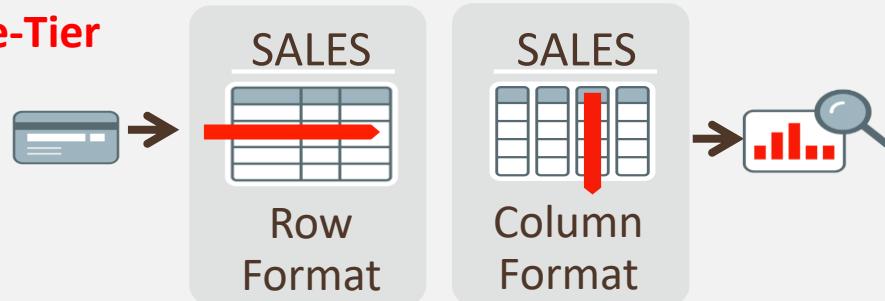
Application-Tier



- TimesTen-In-Memory Database

- Latency Critical OLTP Applications
- **Microsecond** response time
- Standalone or Cache for Oracle Database

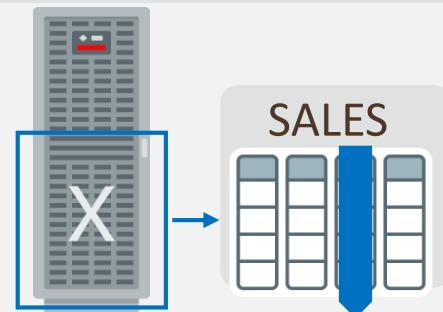
Database-Tier



- Database In-Memory

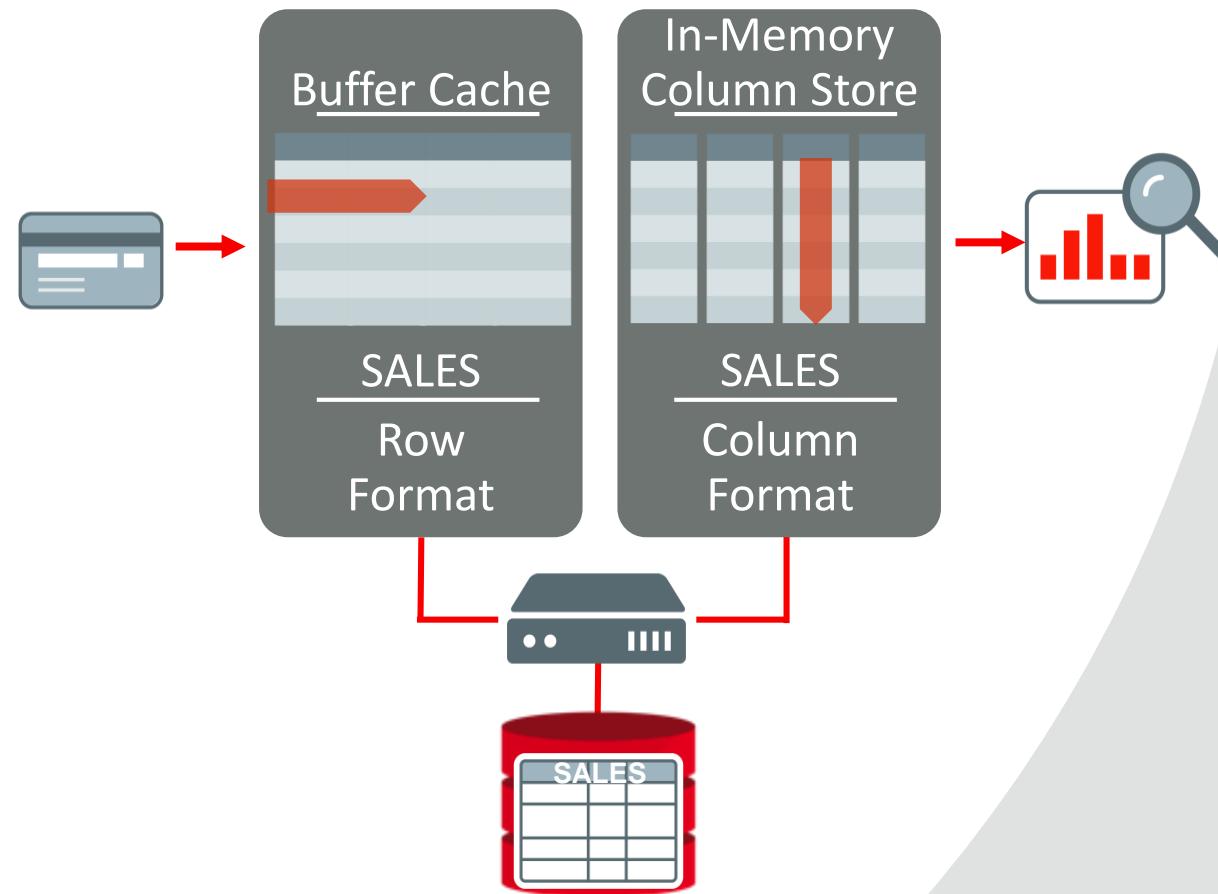
- Dual Format In-Memory Database
- **Billions of Rows/sec** analytic data processing
- **2-3x** Faster Mixed Workloads

Storage-Tier



- In-Memory on Exadata Storage

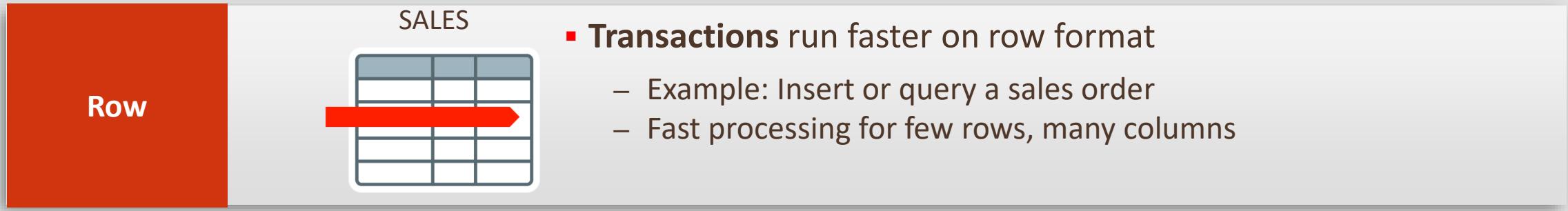
- In-memory format on Exadata Flash Cache
- **5-10x** faster smart scan in storage
- **15x** increase in total columnar capacity



Oracle Database In-Memory

Background

In-Memory Row Format: Slower for Analytics



Buffer Cache

COL1	COL2	COL3	COL4
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X

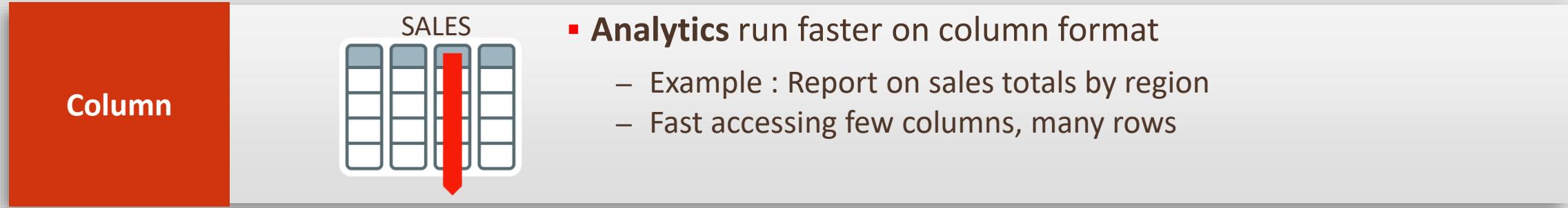
Row Format

`SELECT COL4 FROM MYTABLE;`



Needs to skip over unneeded data

In-Memory Columnar Format: Faster for Analytics



- **Analytics** run faster on column format
 - Example : Report on sales totals by region
 - Fast accessing few columns, many rows

IM Column Store

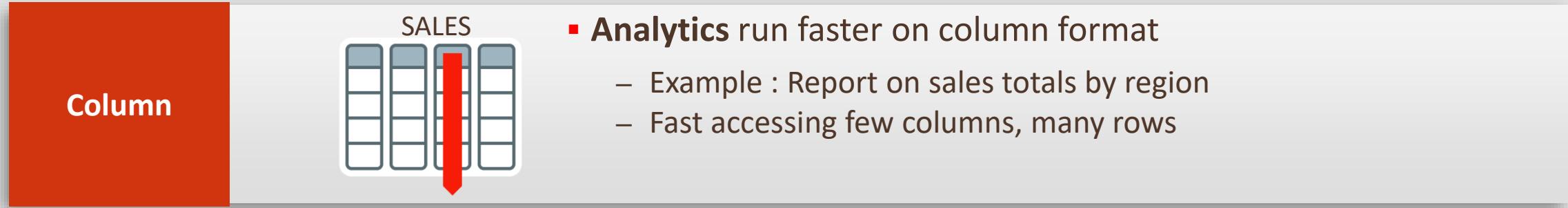
COL1	COL2	COL3	COL4
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X

Column Format

SELECT COL4 FROM MYTABLE ;



In-Memory Columnar Format: Faster for Analytics



IM Column Store

COL1	COL2	COL3	COL4
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X

SELECT **COL4** FROM MYTABLE ;



X X X X

Scans only the data required by the query

Background | Row vs. Column Databases

Row



- **Transactions** run faster on row format
 - Example: Insert or query a sales order
 - Fast processing for few rows, many columns

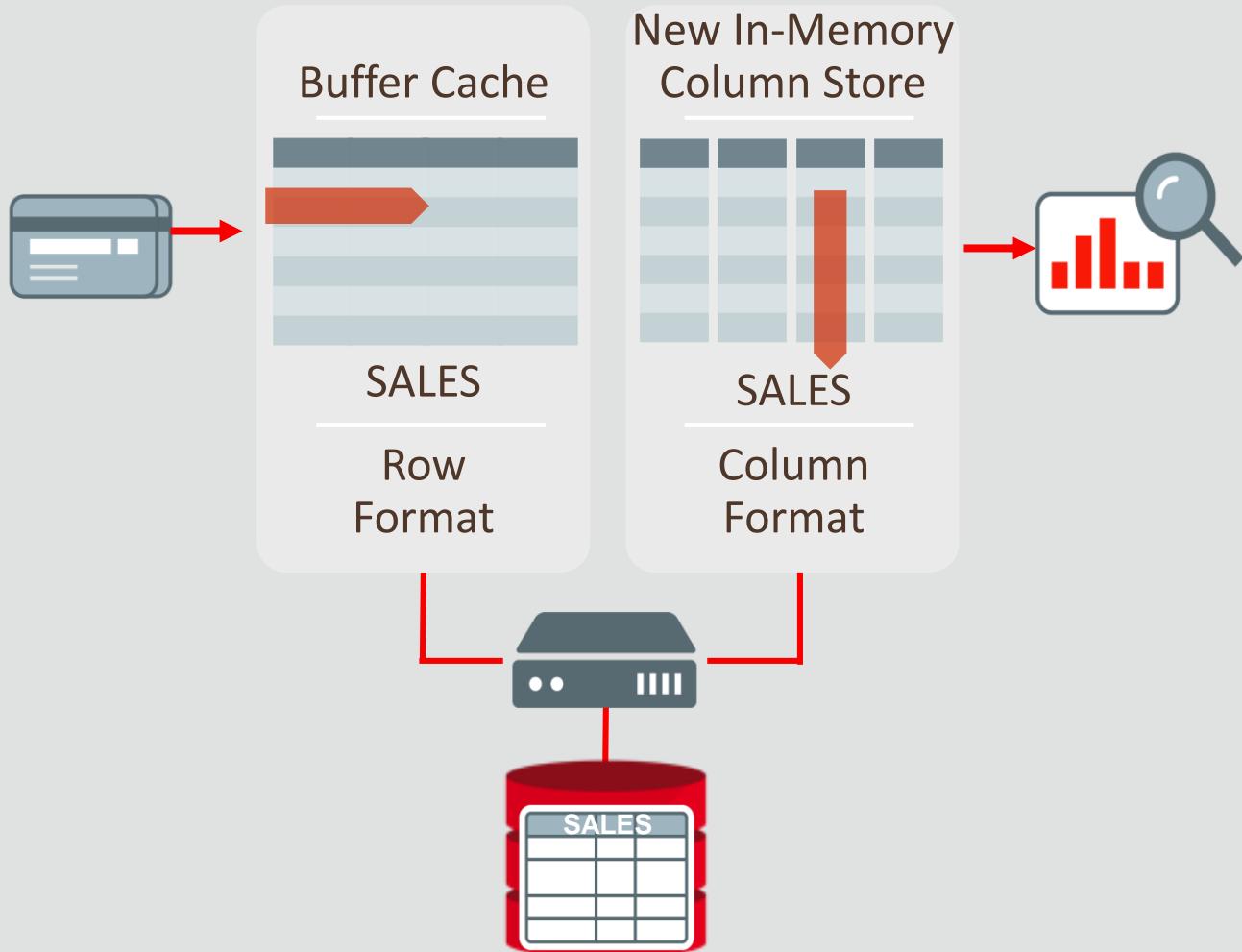
Column



- **Analytics** run faster on column format
 - Example : Report on sales totals by region
 - Fast accessing few columns, many rows

Choose One Format and Suffer the Consequences / Tradeoffs

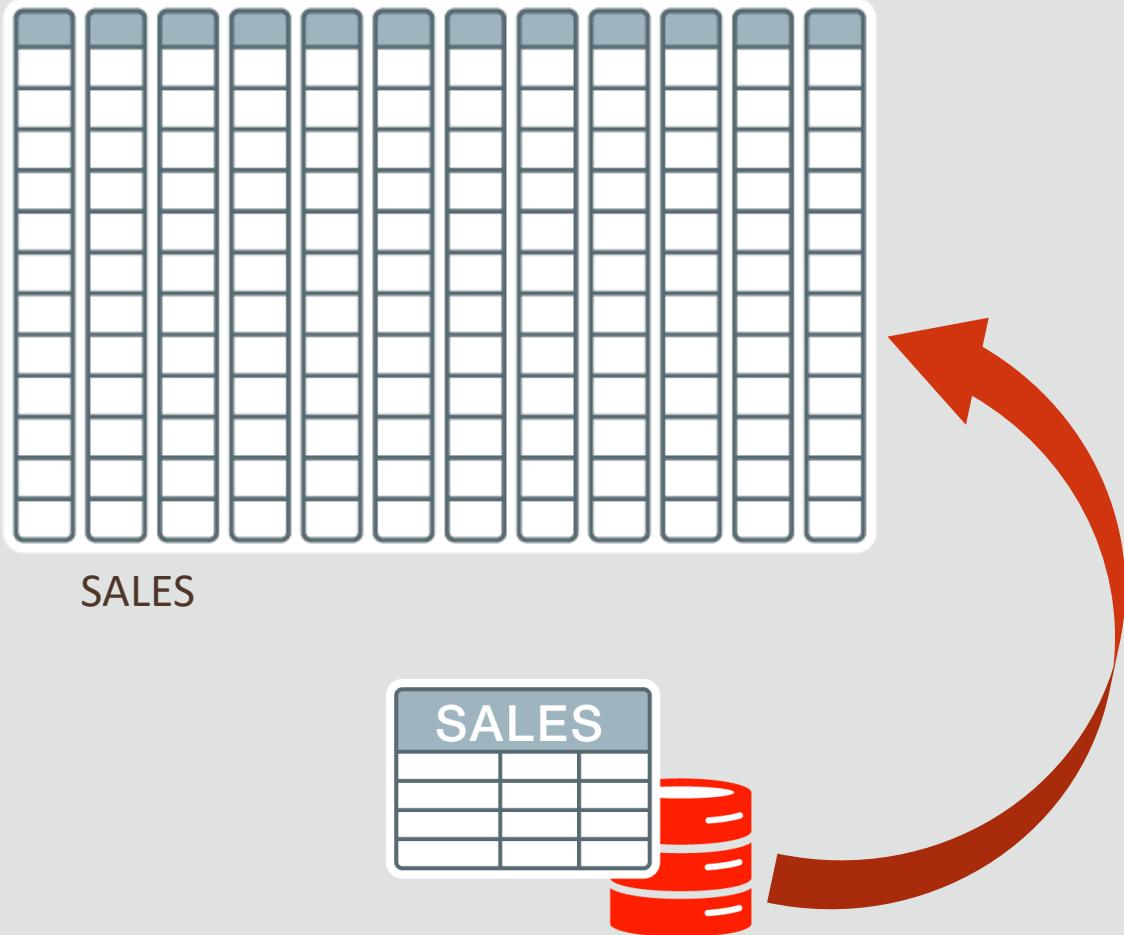
Database In-Memory | Architecture



- Both row and column format for same table
 - Simultaneously active and consistent
- OLTP uses existing row format
- Analytics uses In-Memory Column format
 - Seamlessly built into Oracle Database
 - All enterprise features work - RAC, Dataguard, Flashback, etc.

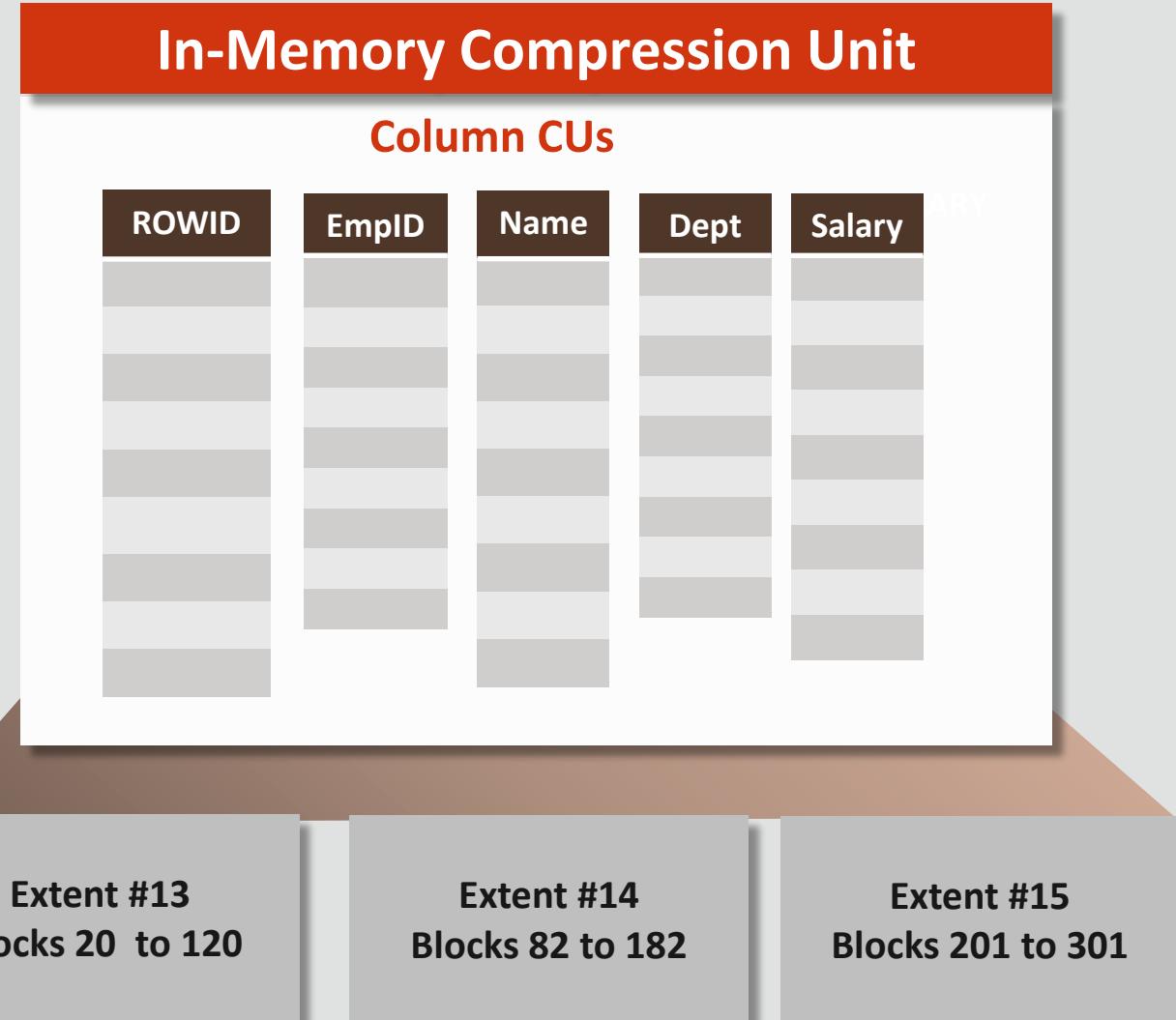
In-Memory Columnar Format

Pure In-Memory Columnar



- Pure in-memory column format
- Fast In-Memory Maintenance with OLTP
- No Changes to Disk Format
- Available on All Platforms
- Enabled at tablespace, table, partition, sub-partition, and even column level
- Total memory area controlled by **inmemory_size** parameter

In-Memory Columnar Format | Deep Dive



In-Memory Compression Unit (IMCU)

- Unit of column store allocation
Spans large number of rows (e.g. 0.5 million) on one or more table extents
 - Each column stored as **Column Compression Unit** (column CU)

Multiple **MEMCOMPRESS** levels:

FOR QUERY – fastest queries

FOR CAPACITY – best compression

In-Memory Columnar Format | Compression

Uncompressed Data

CAT
CAT
FISH
FISH
HORSE
HORSE
HORSE
DOG
DOG
CAT
CAT
FISH
HORSE
HORSE
DOG
DOG



In-Memory Columnar Format | Compression

Uncompressed Data

CAT
CAT
FISH
FISH
HORSE
HORSE
HORSE
DOG
DOG
CAT
CAT
FISH
HORSE
HORSE
DOG
DOG



Dictionary Compressed

CAT	0
DOG	1
FISH	2
HORSE	3
	00, 00, 10, 10
	11, 11, 11, 01
	01, 00, 00, 10
	11, 11, 01, 01



In-Memory Columnar Format | Compression

Uncompressed Data
CAT
CAT
FISH
FISH
HORSE
HORSE
HORSE
DOG
DOG
CAT
CAT
FISH
HORSE
HORSE
DOG
DOG



Dictionary Compressed

CAT	0
DOG	1
FISH	2
HORSE	3
	00, 00, 10, 10
	11, 11, 11, 01
	01, 00, 00, 10
	11, 11, 01, 01

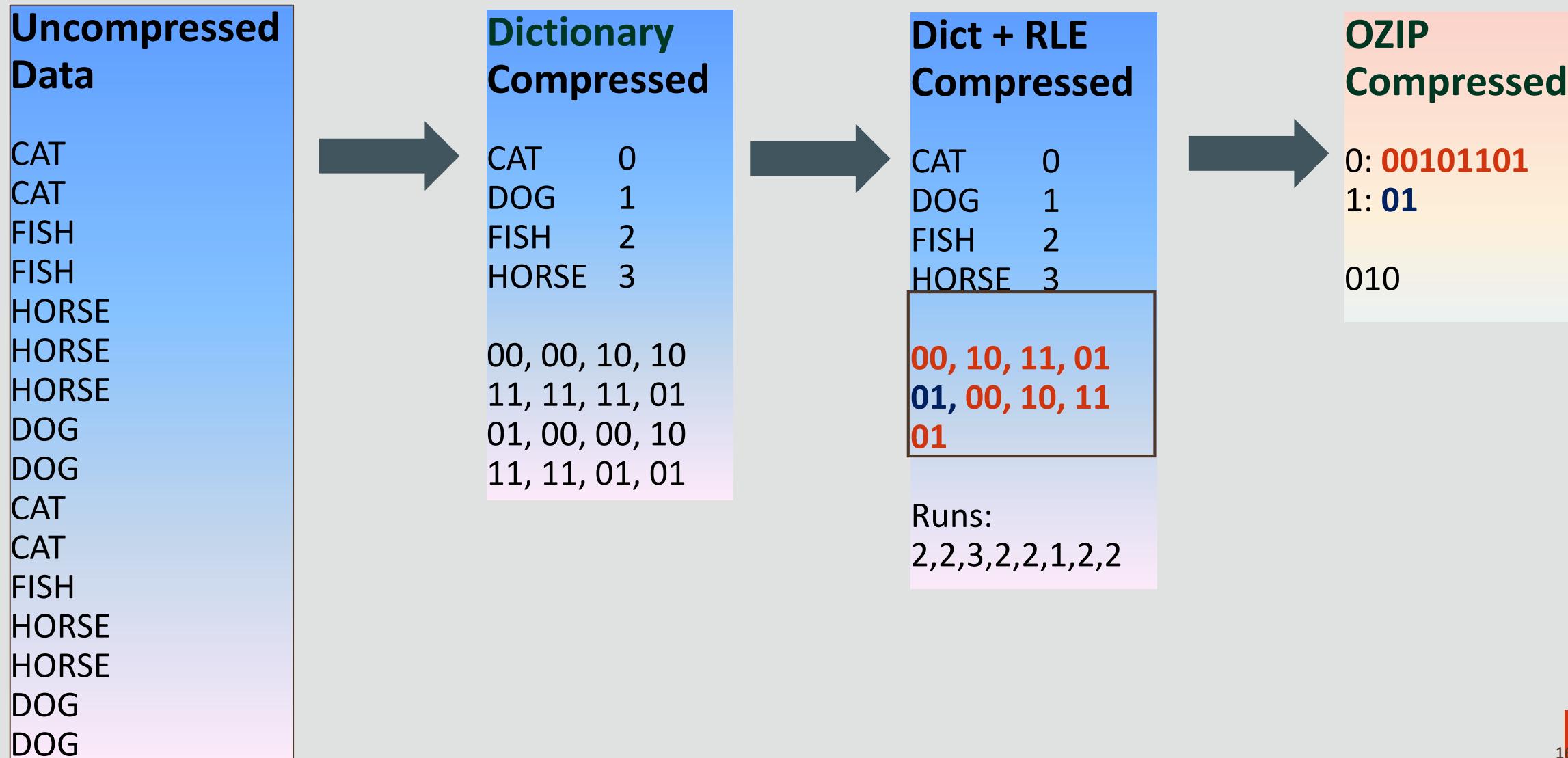


Dict + RLE Compressed

CAT	0
DOG	1
FISH	2
HORSE	3
	00, 10, 11, 01
	01, 00, 10, 11
	01

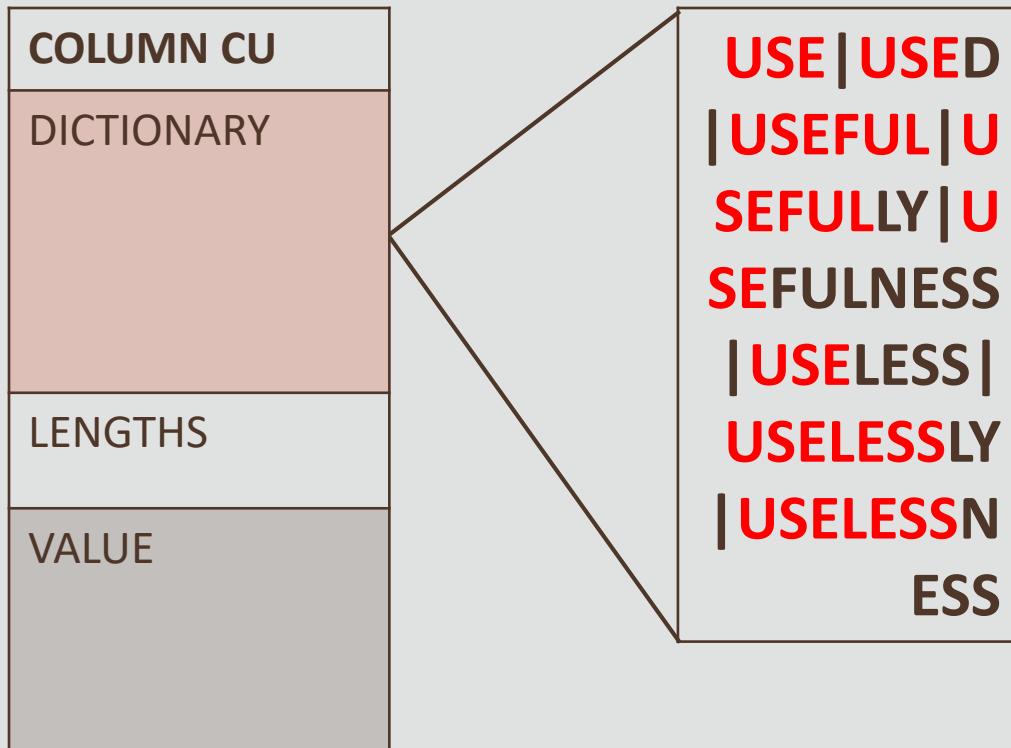
Runs:
2,2,3,2,2,1,2,2

In-Memory Columnar Format | Compression

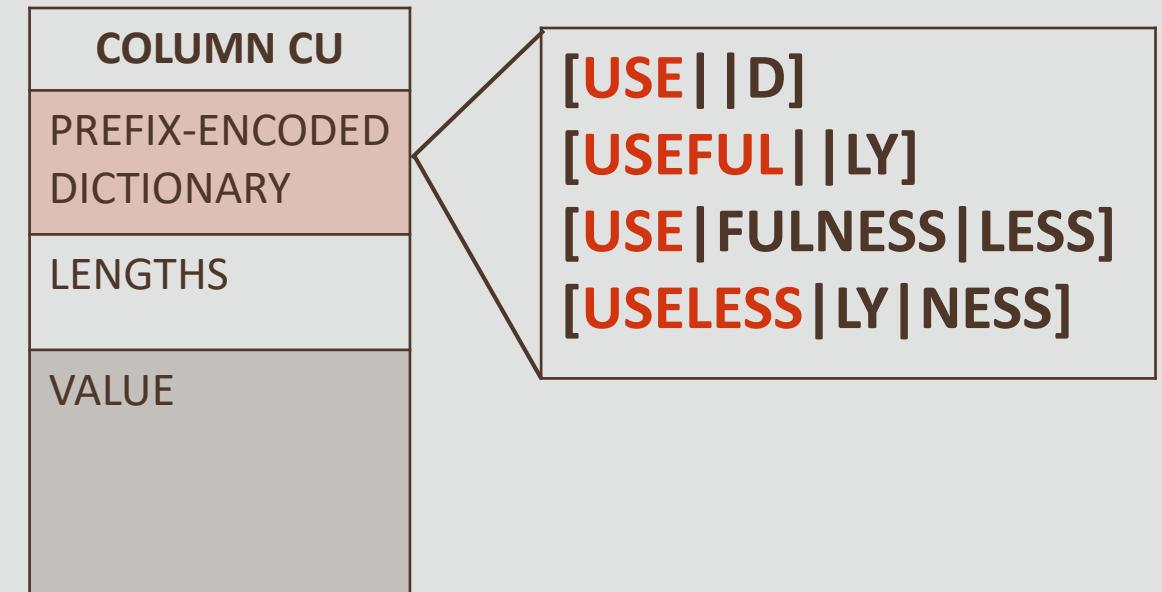


In-Memory Columnar Format | Compression

No Compression

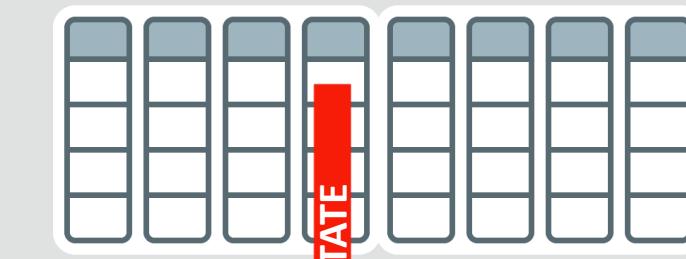


Prefix Compression



In-Memory Enables SIMD Vector Processing

Memory



> 100x Faster

Example:
Find sales in
State of California

- Column format benefit: Need to access only needed columns
- Process multiple values with a single SIMD Vector Instruction
- **Billions of rows/sec** scan rate per CPU core
 - Row format is millions/sec

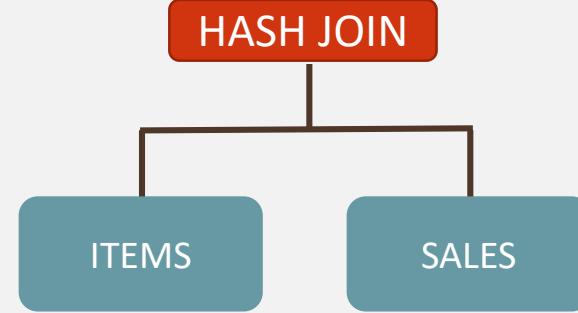
Improves All Aspects of Analytic Workloads...

Scans



- Billions of Rows per second scans

Joins

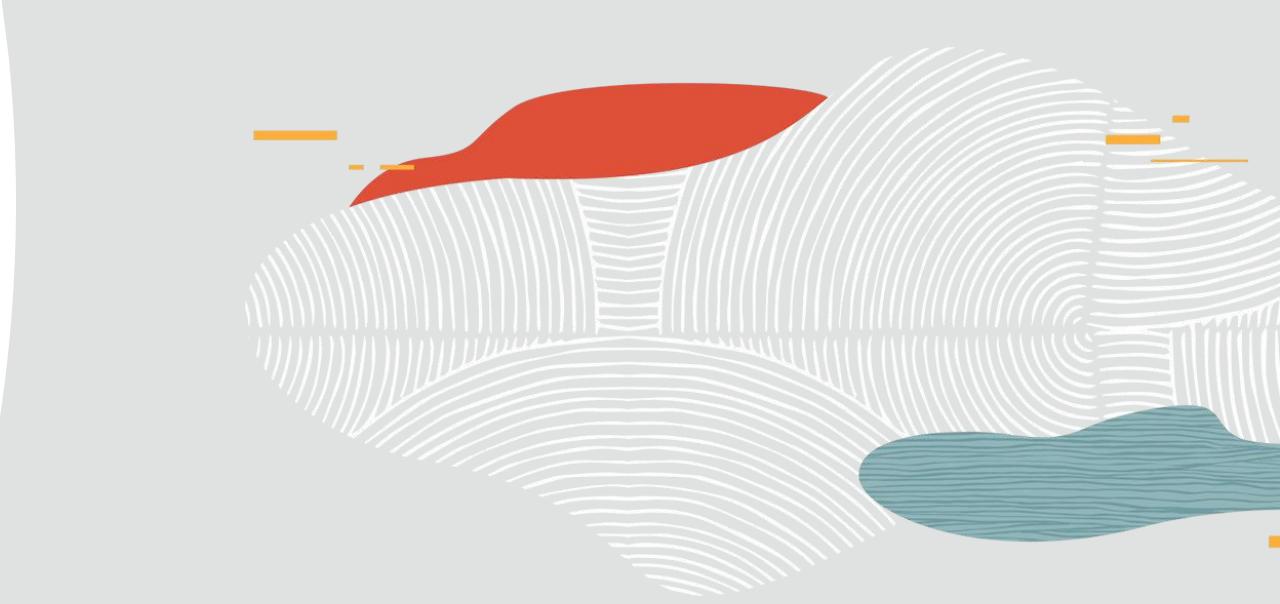
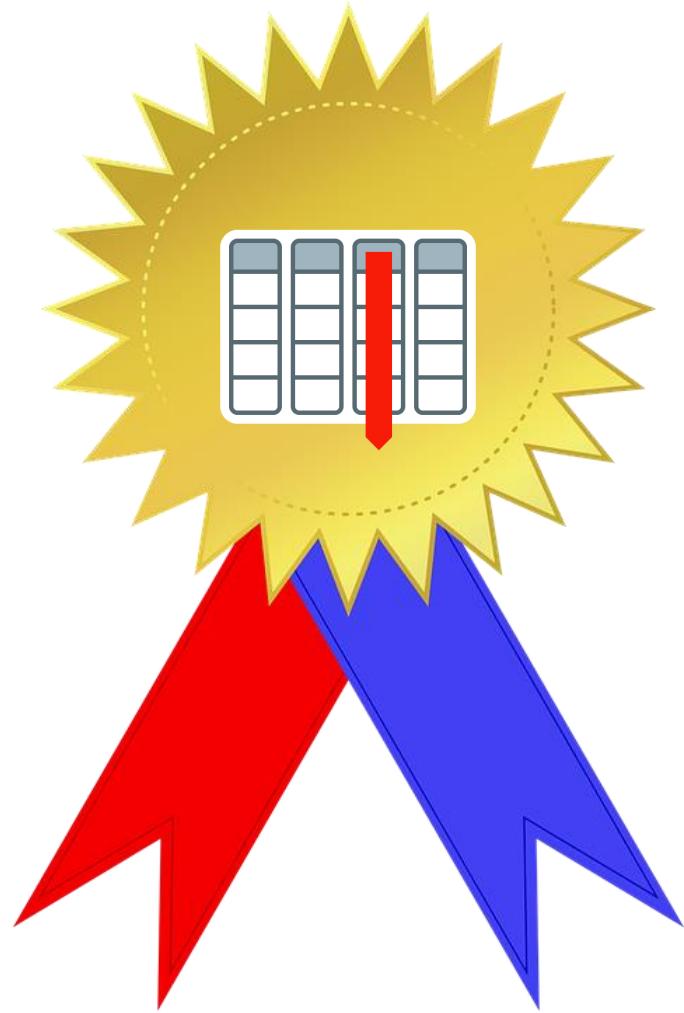


- Convert slower joins into 10x faster filtered column scans

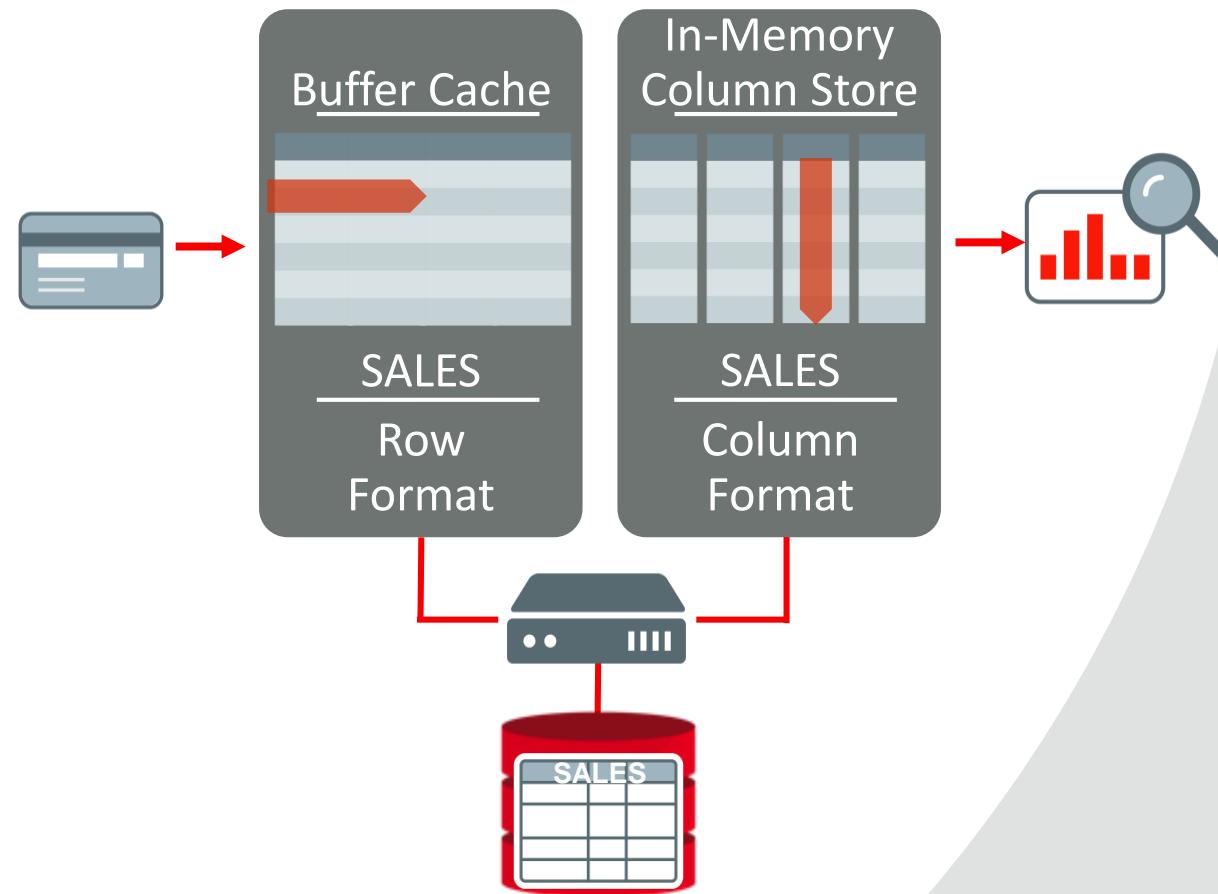
Reporting



- Run reports with aggregations and joins 10x faster



Top-5 Oracle Database In-Memory Innovations



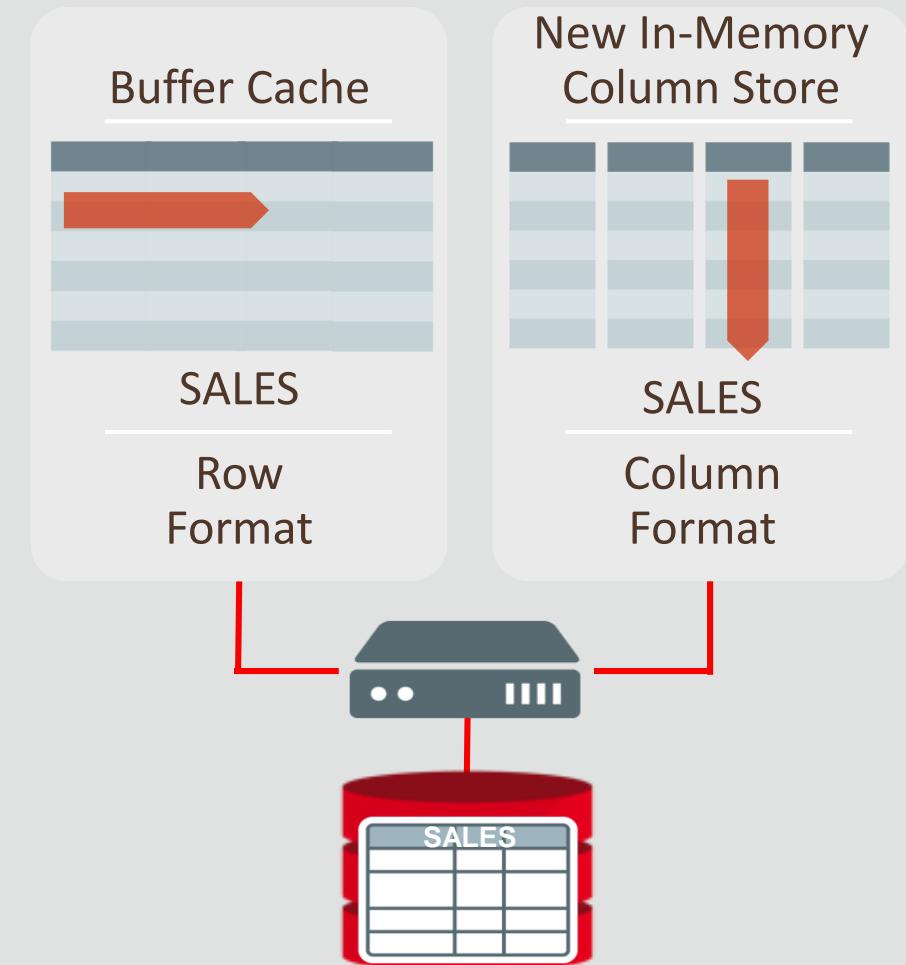
#1

Dual-Format Architecture

*Fast Mixed Workloads, Faster
Analytics*

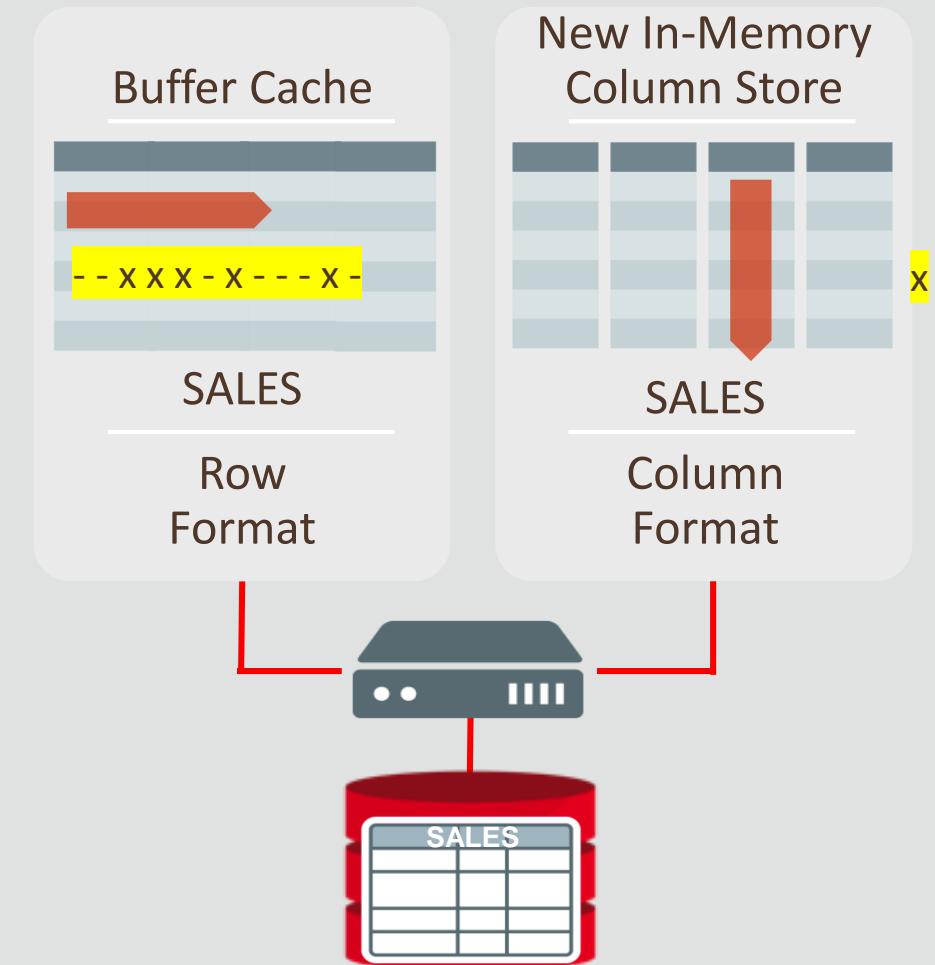
In-Memory: Dual-Format Architecture

- Dual-Format Architecture enables fast Mixed Workloads and faster Analytics
- Fast In-Memory DML because invalid row is logically removed from column store (just set a bit)



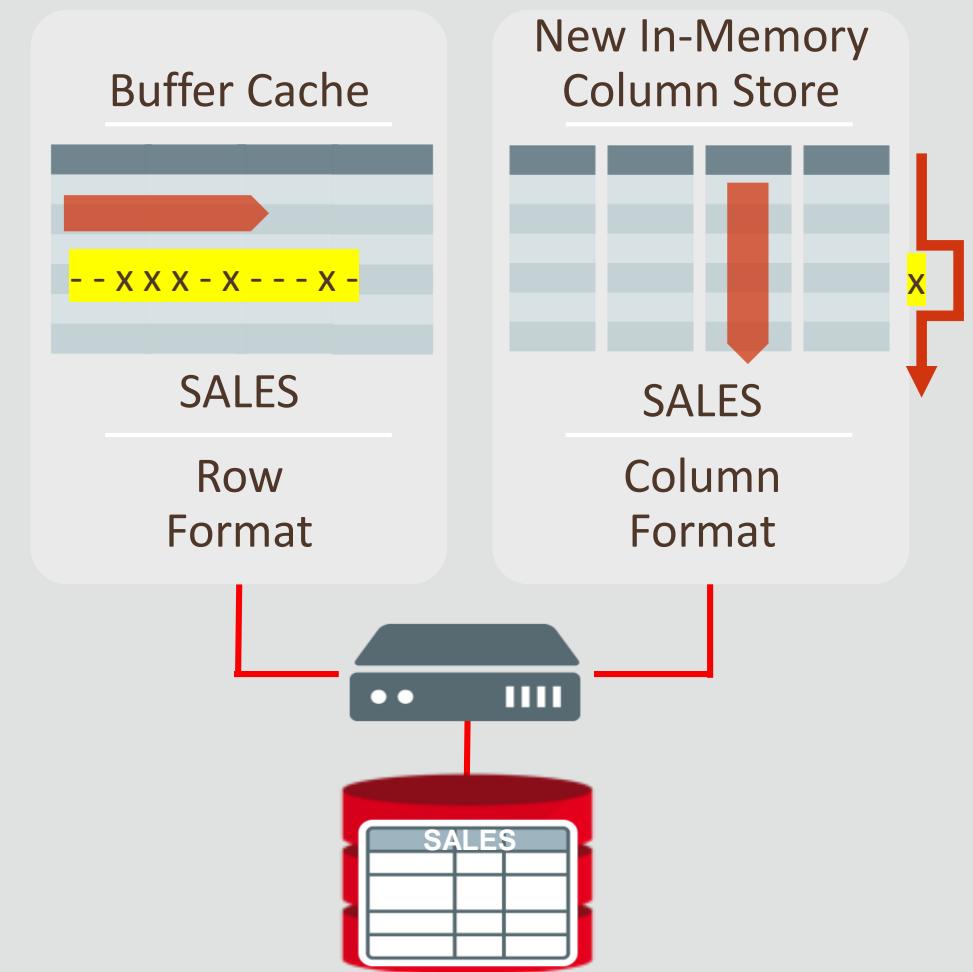
In-Memory: Dual-Format Architecture

- Dual-Format Architecture enables fast Mixed Workloads and faster Analytics
- Fast In-Memory DML because invalid row is logically removed from column store (just set a bit)



In-Memory: Dual-Format Architecture

- Dual-Format Architecture enables fast Mixed Workloads and faster Analytics
- Fast In-Memory DML because invalid row is logically removed from column store (just set a bit)
- Analytic query will ignore invalid rows in column store, and just vector process valid rows. Invalid rows are then processed.
 - IMCUs not covering invalid rows are unaffected.
- Mixed workload performance can suffer if the number of invalid rows accumulates in IMCUs
 - **Fast repopulation techniques save the day!**



In-Memory: Fast Background Repopulation

Continuous intelligence to track how dirty an IMCU is, how frequently it is scanned, and when to take action to refresh/repopulate it.

Double Buffering

THEN

NOW



Old IMCU stays online until New IMCU is built. Then A *switcheroo* happens once New IMCU is ready.

Incremental Repopulation



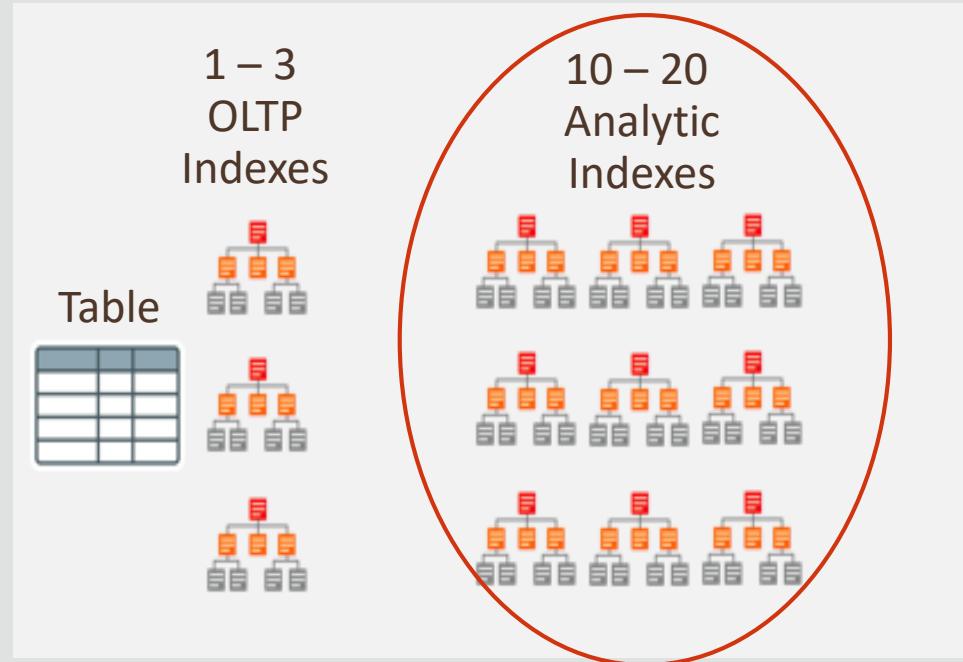
Build new columns in new IMCU using meta-data present in the old IMCU, allowing quick formatting

Column-Level Invalidations

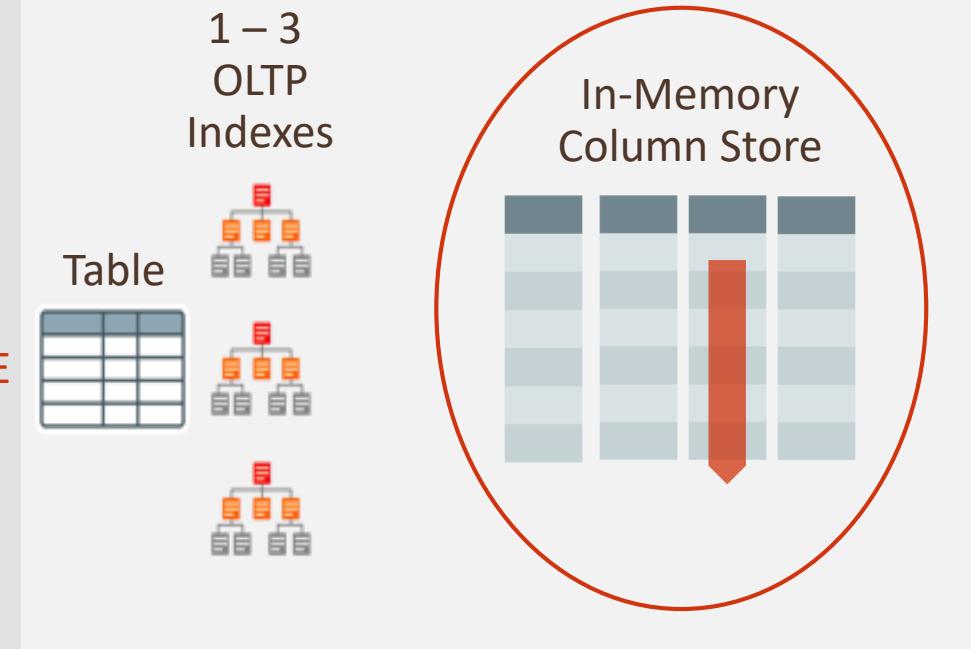


Column-Level Invalidations tracked to still enable IM scans

Accelerates Mixed Workloads (Hybrid OLTP)

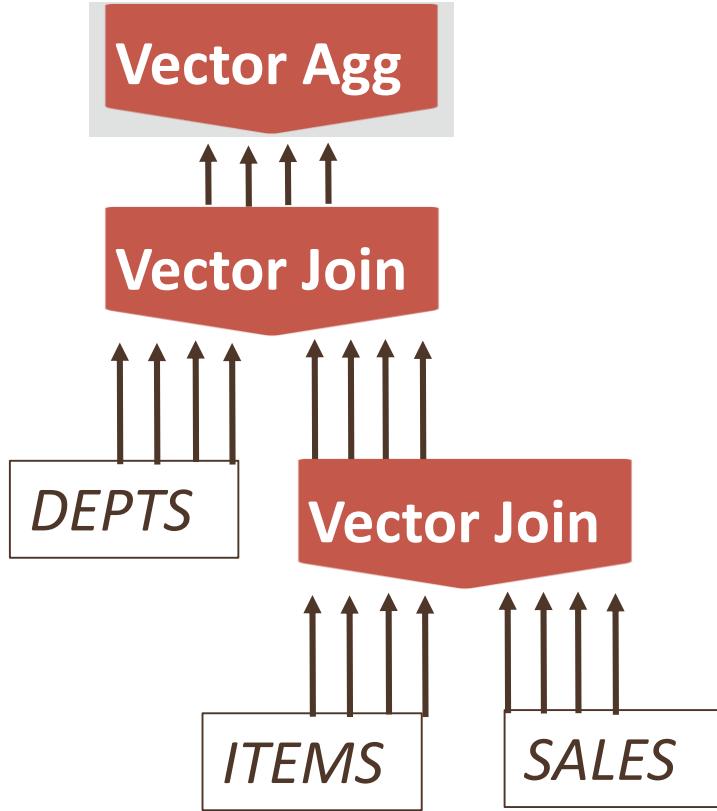


REPLACE



- Inserting one row into a table requires updating 10-20 analytic indexes: **Slow!**
- Fast analytics only on indexed columns
- Analytic indexes **increase** database size

- Column Store not persistent so updates are: **Fast!**
- Fast analytics on any columns
- No analytic indexes: **Reduces** database size



#2

Vectorized Analytics

*SIMD Vector Processing at Billions
of Rows per Second*

Faster Scans | SIMD Vector Processing

- Parallelize predicate evaluation – load, eval, store/consume result
- Select count(*) from T where $a > 10$ and $b < 20$
 - [Load] A

96	64	32	0
51	95182	1	69

Faster Scans | SIMD Vector Processing

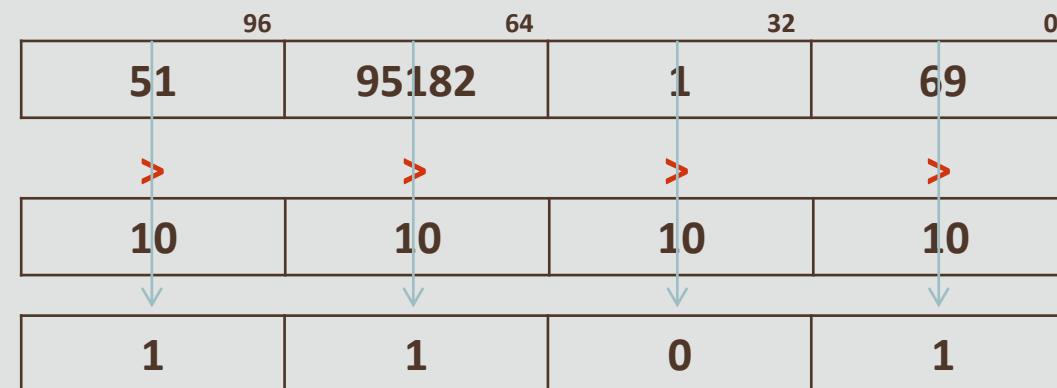
- Parallelize predicate evaluation – load, eval, store/consume result
- Select count(*) from T where $a > 10$ and $b < 20$
 - [Load] A
 - [Load] Temp = 10

96	64	32	0
51	95182	1	69

10	10	10	10
----	----	----	----

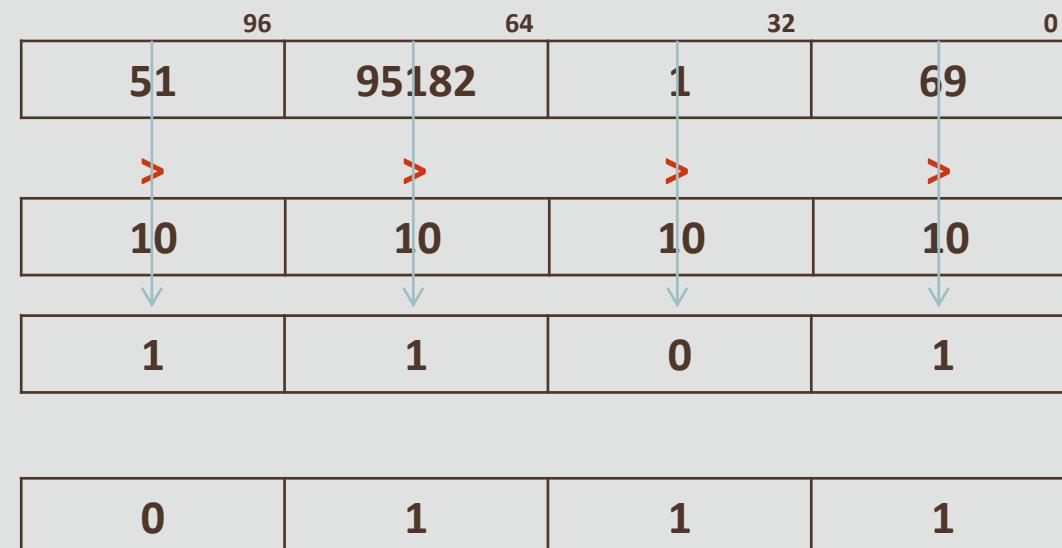
Faster Scans | SIMD Vector Processing

- Parallelize predicate evaluation – load, eval, store/consume result
- Select count(*) from T where $a > 10$ and $b < 20$
 - [Load] A
 - [Load] Temp = 10
 - [Compare] A > Temp



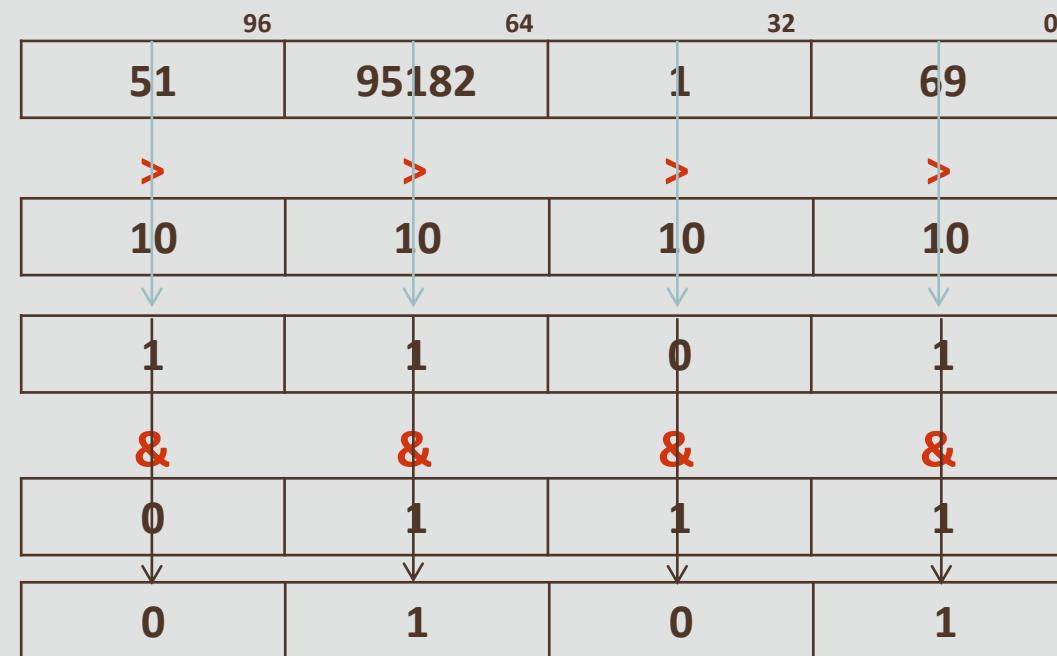
Faster Scans | SIMD Vector Processing

- Parallelize predicate evaluation – load, eval, store/consume result
- Select count(*) from T where $a > 10$ and $b < 20$
 - [Load] A
 - [Load] Temp = 10
 - [Compare] A > Temp
- Load B, Compare 20



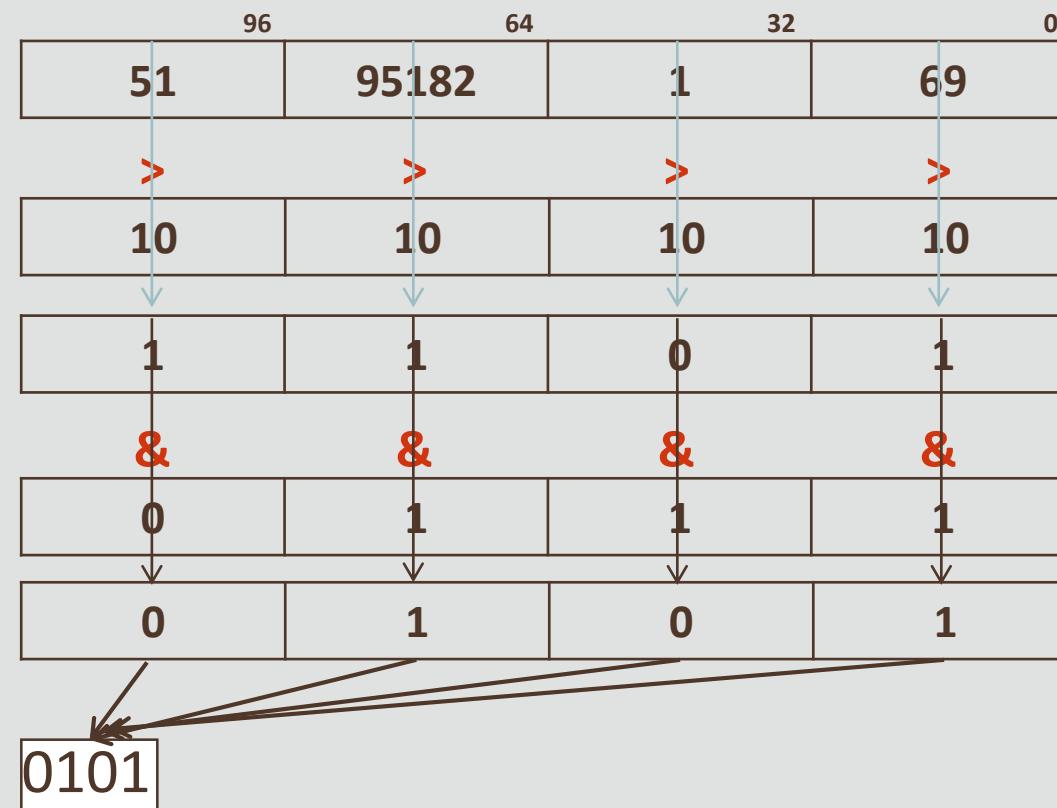
Faster Scans | SIMD Vector Processing

- Parallelize predicate evaluation – load, eval, store/consume result
- Select count(*) from T where $a > 10$ and $b < 20$
 - [Load] A
 - [Load] Temp = 10
 - [Compare] A > Temp
- Load B, Compare 20
- And



Faster Scans | SIMD Vector Processing

- Parallelize predicate evaluation – load, eval, store/consume result
- Select count(*) from T where $a > 10$ and $b < 20$
 - [Load] A
 - [Load] Temp = 10
 - [Compare] A > Temp
- Load B, Compare 20
- And
- Mask, Store Bit-Map



Faster Analytics | In-Memory Joins

Example: Find sales price of each Vehicle



```
CREATE INMEMORY JOIN GROUP v_name_jg  
(VEHICLES (NAME) ,SALES (NAME)) ;
```

- Joins are a significant component of analytic queries
 - Joins on inmemory tables are 10x faster already
- **Join Groups** enables faster joins
 - Specifies columns used to join tables
 - Join columns compressed using exact same encoding scheme.
 - This enables a faster array-based indexing join to be used instead of expensive hash join.
- Enables **2-3x speedup** over already fast inmemory joins

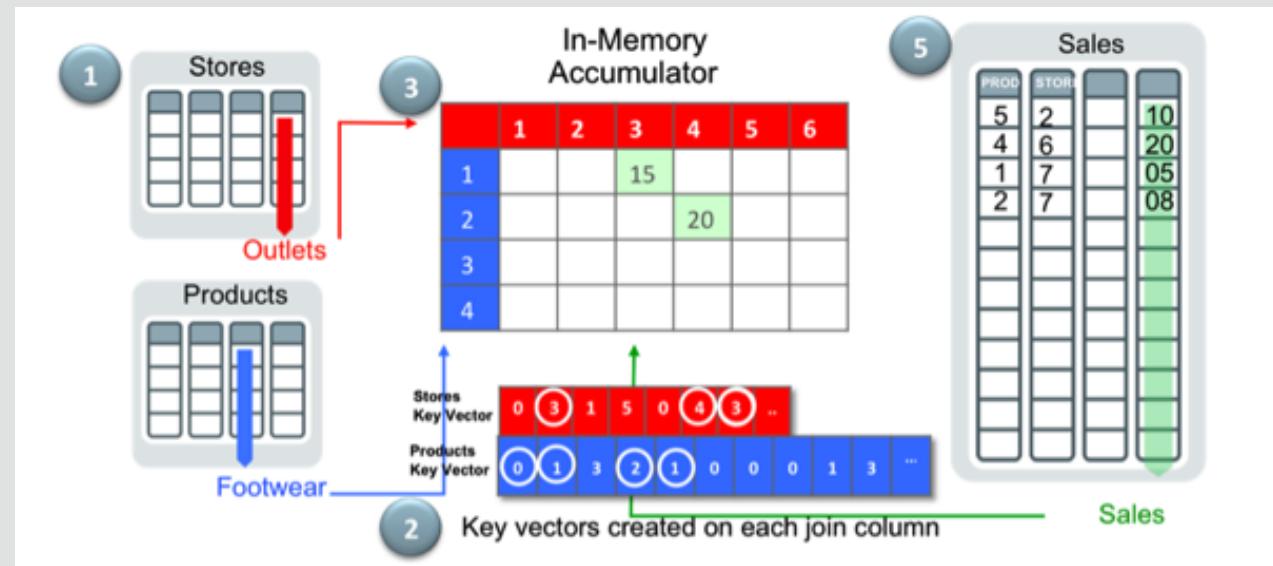
Faster Analytics | In-Memory Aggregation

Aggregation Push-Down

- Improve Single Table Aggregation
- Push aggregation operators down into the scan operators
- Reduce number of rows flowing back up to SQL layer
- New aggregation algorithms leveraging In-Memory data formats and SIMD
- **2-10X improvements**

Vector Transformation

- Improve Aggregation over Joins
- Query transformation replaces aggregation over hash-joins with new push-down operators.



GBY Sum for VERY LARGE Numbers

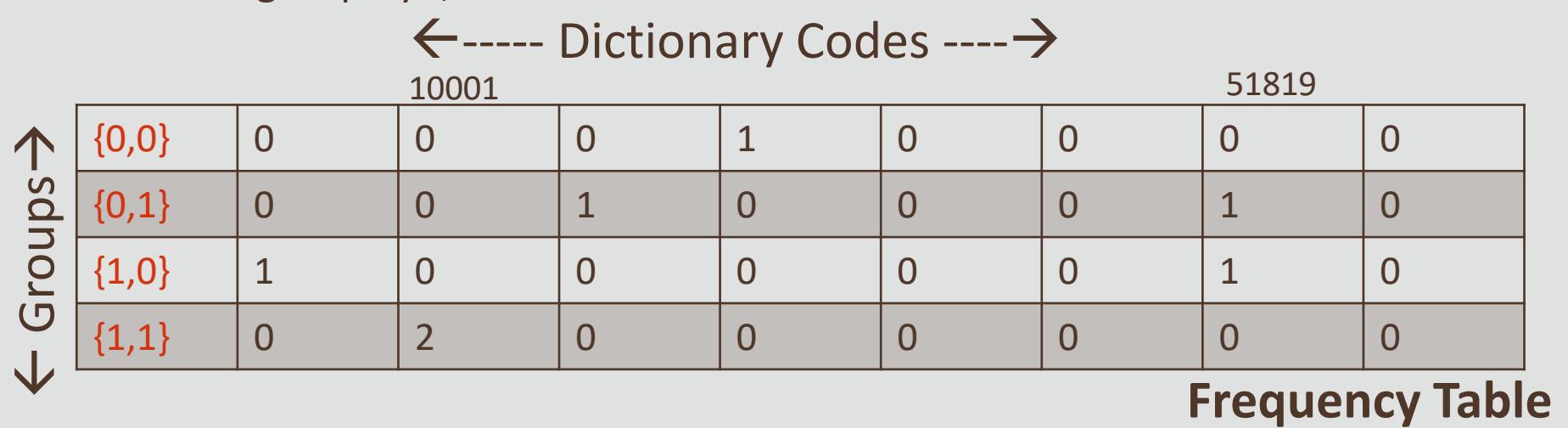
- “Select sum(A) from T where ... group by J, K”

A	J	K
51819	0	1
23591	0	1
51819	1	0
39510	0	0
10001	1	1
9599	1	0
10001	1	1

GBY Sum for VERY LARGE Numbers

- “Select sum(A) from T where ... group by J, K”

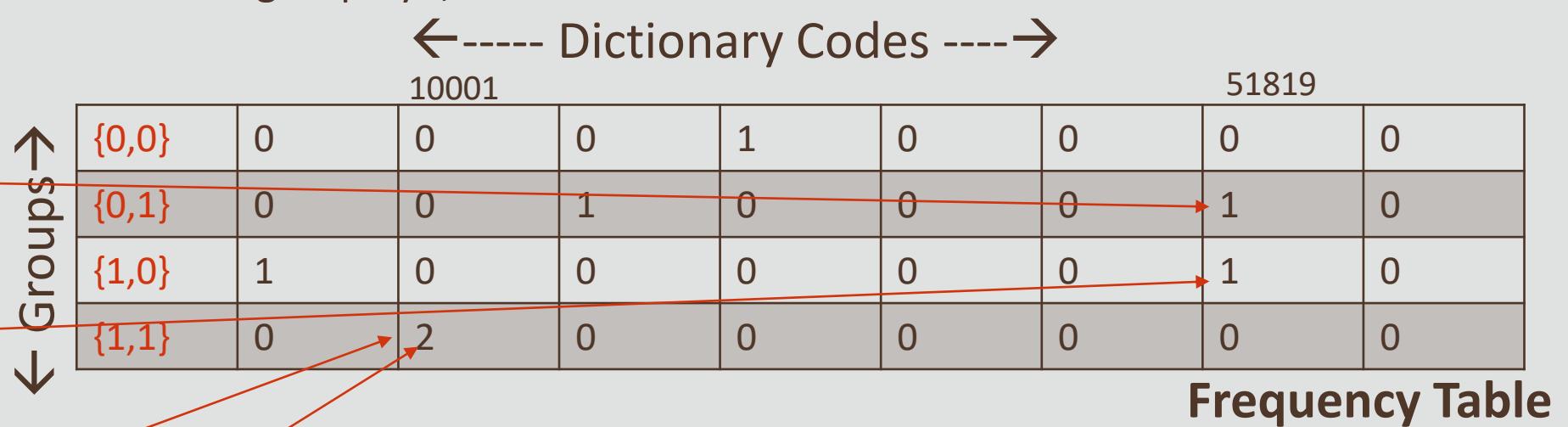
A	J	K
51819	0	1
23591	0	1
51819	1	0
39510	0	0
10001	1	1
9599	1	0
10001	1	1



GBY Sum for VERY LARGE Numbers

- “Select sum(A) from T where ... group by J, K”

A	J	K
51819	0	1
23591	0	1
51819	1	0
39510	0	0
10001	1	1
9599	1	0
10001	1	1

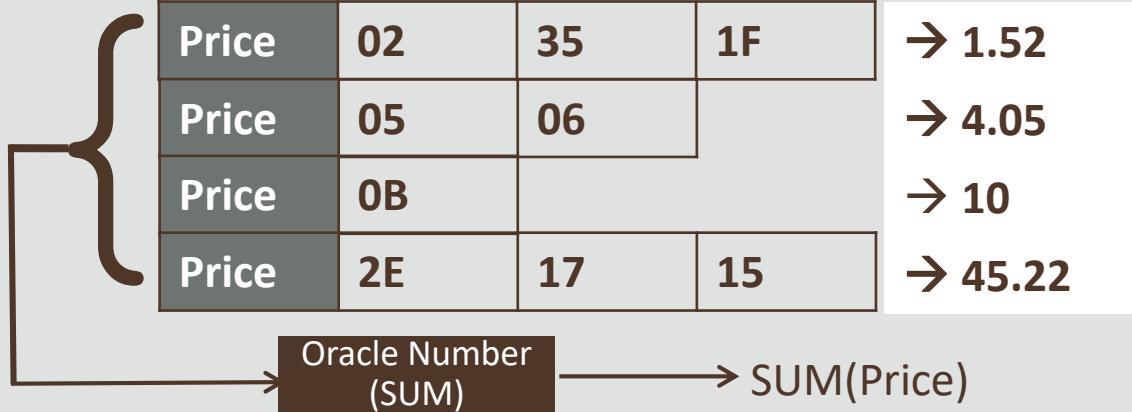


AGG[GROUP_ID] = FREQUENCY * DICT_SYMBOL

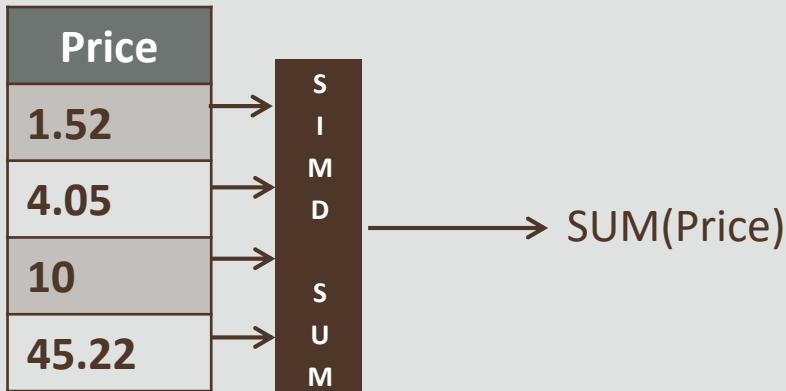
$$\begin{aligned}
 \text{AGG}\{0,0\} &= (0 * \text{DICT}[0]) + (0 * \text{DICT}[1]) + \dots + (1 * \text{DICT}[39510]) + \dots \\
 \text{AGG}\{0,1\} &= (0 * \text{DICT}[0]) + \dots + (1 * \text{DICT}[23591]) + \dots + (1 * \text{DICT}[51819]) + \dots \\
 \text{AGG}\{1,0\} &= (0 * \text{DICT}[0]) + \dots + (1 * \text{DICT}[9599]) + \dots + (1 * \text{DICT}[51819]) + \dots \\
 \text{AGG}\{1,1\} &= (0 * \text{DICT}[0]) + \dots + (2 * \text{DICT}[10001]) + \dots
 \end{aligned}$$

Faster Analytics | In-Memory Numbers

SLOW Row-by-Row Oracle Number Processing



FAST SIMD Vector Processing of In-Memory Numbers

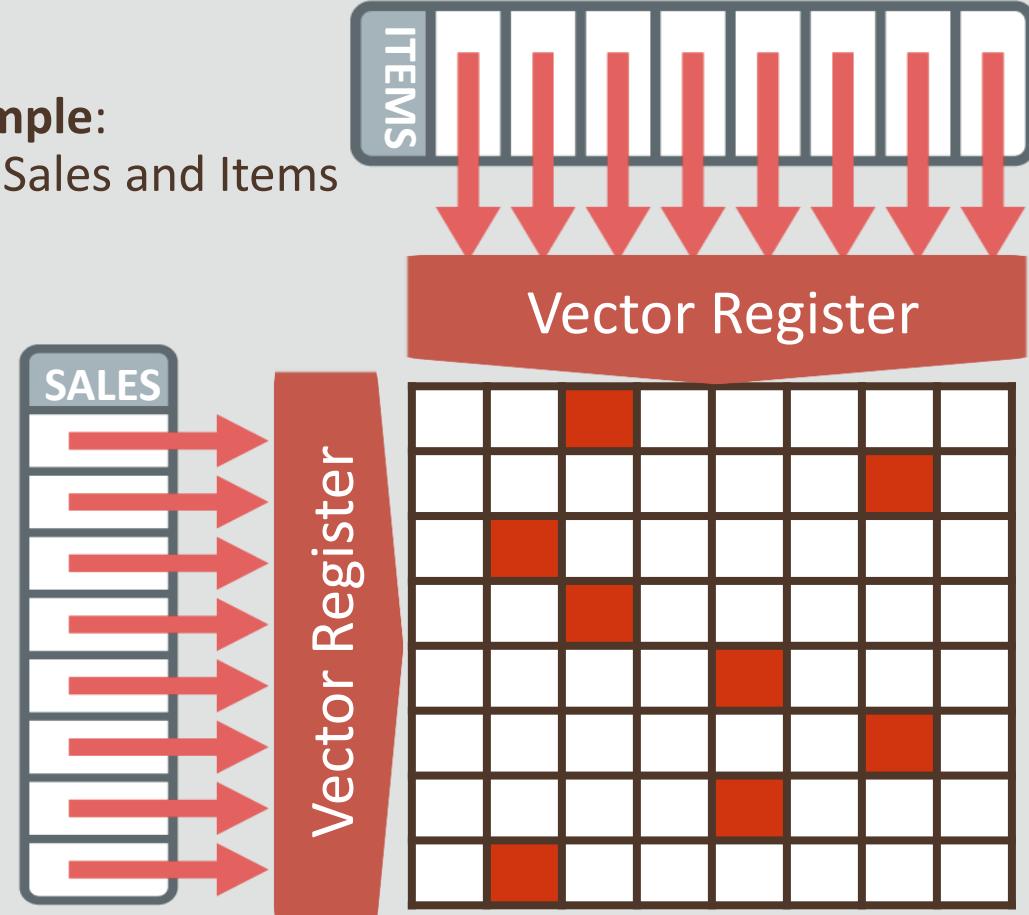


- In-Memory optimized format for NUMBER columns
 - Instead of software-implemented, variable-width ORACLE NUMBERS
 - Enabled using new parameter **inmemory_optimized_arithmetic**
- SIMD Vector Processing on optimized inmemory number format
- Aggregation and Arithmetic operators can improve **up to 20X**

Preview | In-Memory Vector Joins

- New *Deep Vectorization* framework allows SIMD vectorization for a wide range of query operators

Example:
Join Sales and Items

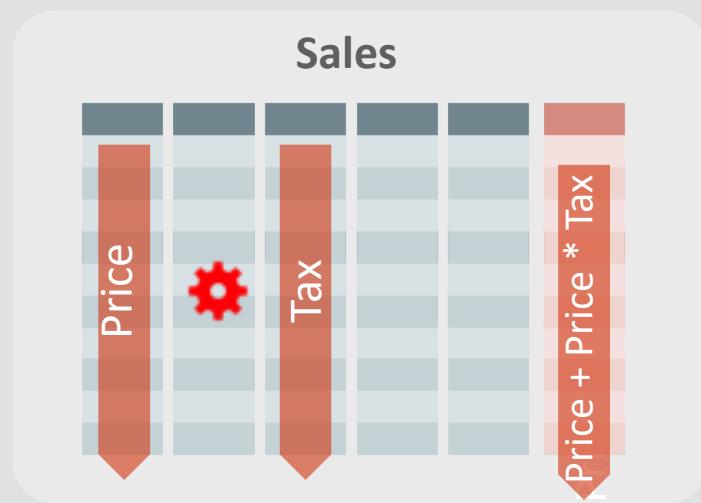


- *In-Memory Vector Joins* uses this framework to accelerate **Complex Joins**
 - Match multiple rows between SALES and ITEMS tables in a single SIMD Vector Instruction
 - 5-10x faster in-memory join processing

Faster Analytics | In-Memory Expressions

Example: Compute total sales price

$$\text{Net} = \text{Price} + \text{Price} * \text{Tax}$$

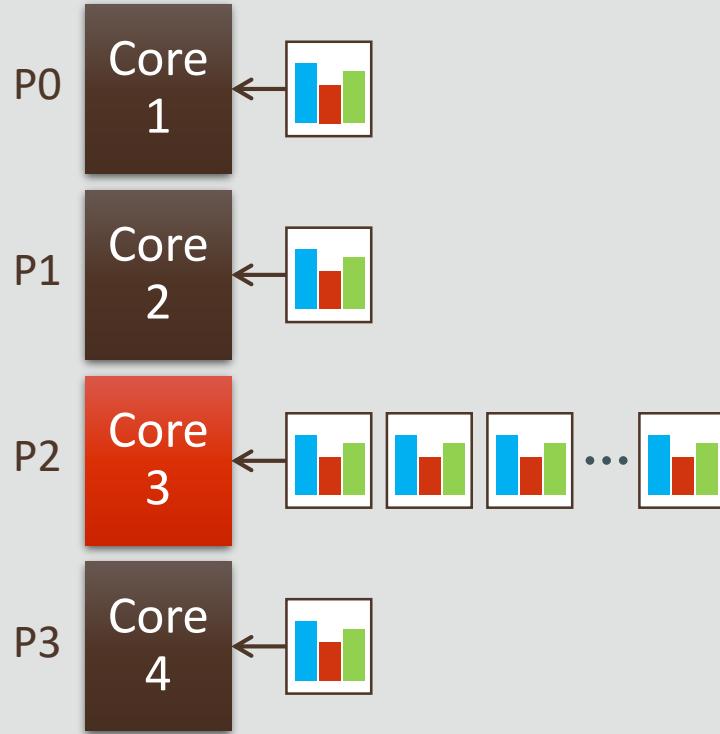


```
CREATE TABLE SALES (
    PRICE NUMBER, TAX NUMBER, ...,
    NET AS (PRICE + PRICE * TAX)
)
INMEMORY;
```

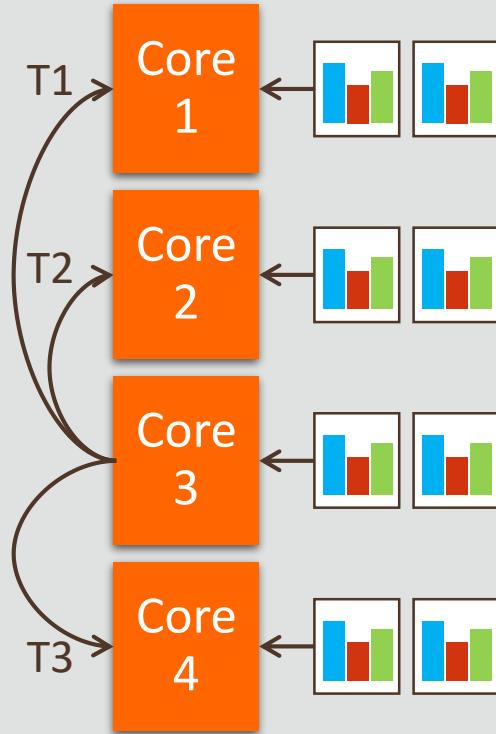
- Hot expressions can be stored as additional columns in memory
- All In-Memory optimizations apply to expression columns (e.g. Vector processing, storage indexes)
- Two modes:
 - **Manual:** Declare virtual columns for desired inmemory expressions
 - **Auto:** Auto detect frequent expressions
- **3-5x** faster complex queries

Faster Analytics | In-Memory Dynamic Scans

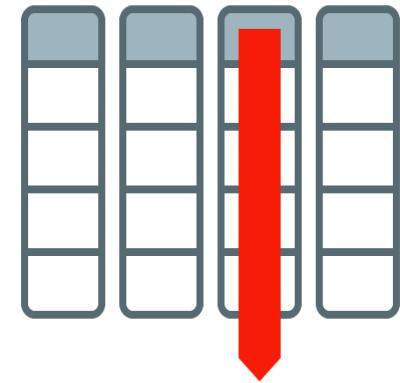
Parallel SQL



Parallel SQL + IMDS



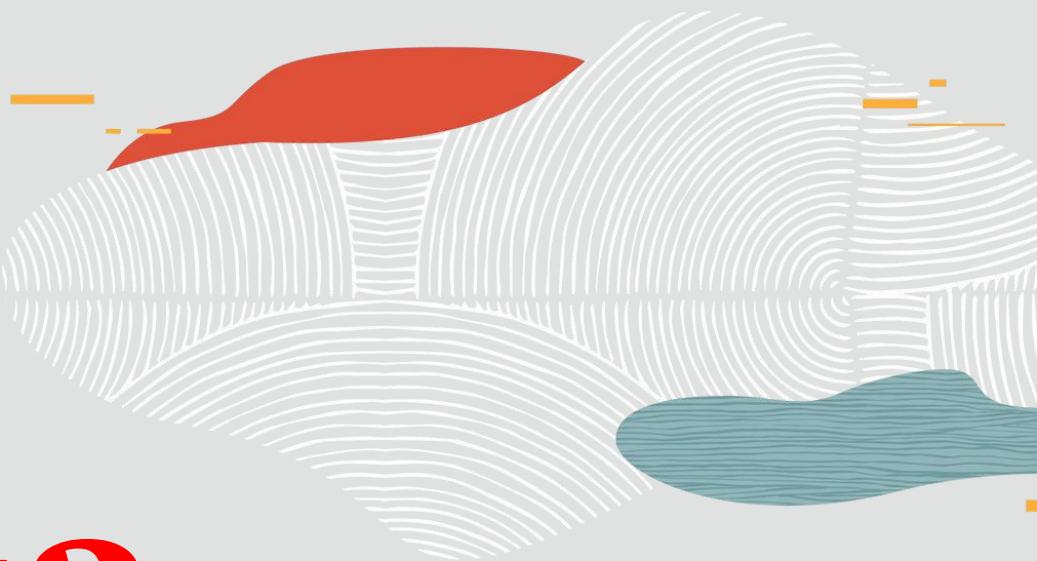
- Parallelize operations pushed down to SCAN layer using light-weight threads
- Supplements *static* PQ plans with faster response times for shorter queries.
 - Achieve PQ execution times for single-threaded queries
- Elastic DOP Rebalancing using Resource Manager
- **Up to 2X gains seen**



#3

In-Memory + Exadata

*In-Flash Columnar Processing at
Cloud Scale*



Background | Exadata Vision

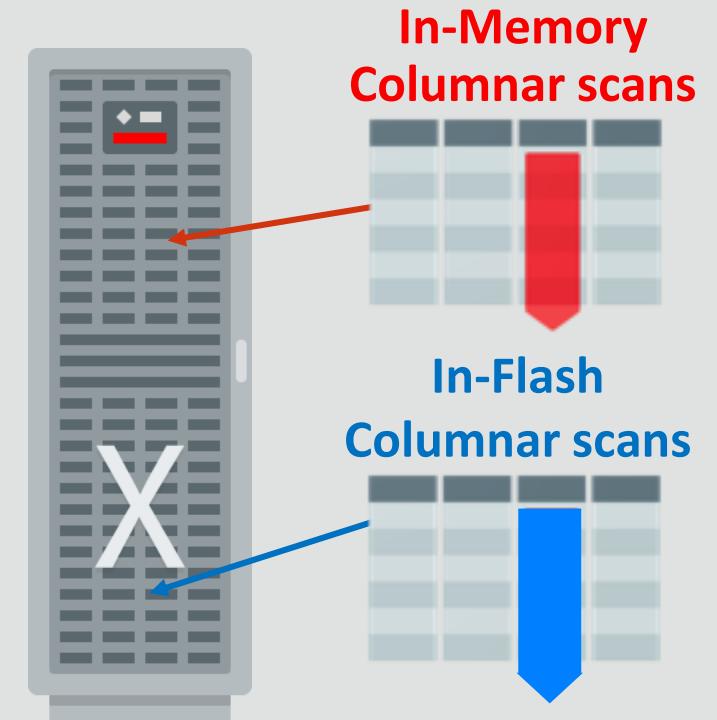
Dramatically Better Platform for All Database Workloads



- **Ideal Database Hardware** – Scale-out, database optimized compute, networking, and storage for fastest performance and lowest cost
- **Smart System Software** – Specialized algorithms vastly improve all aspects of database processing: OLTP, Analytics, Consolidation
- **Automated Management** – Automation and optimization of configuration, updates, performance, and management culminating in Fully Autonomous Infrastructure and Database

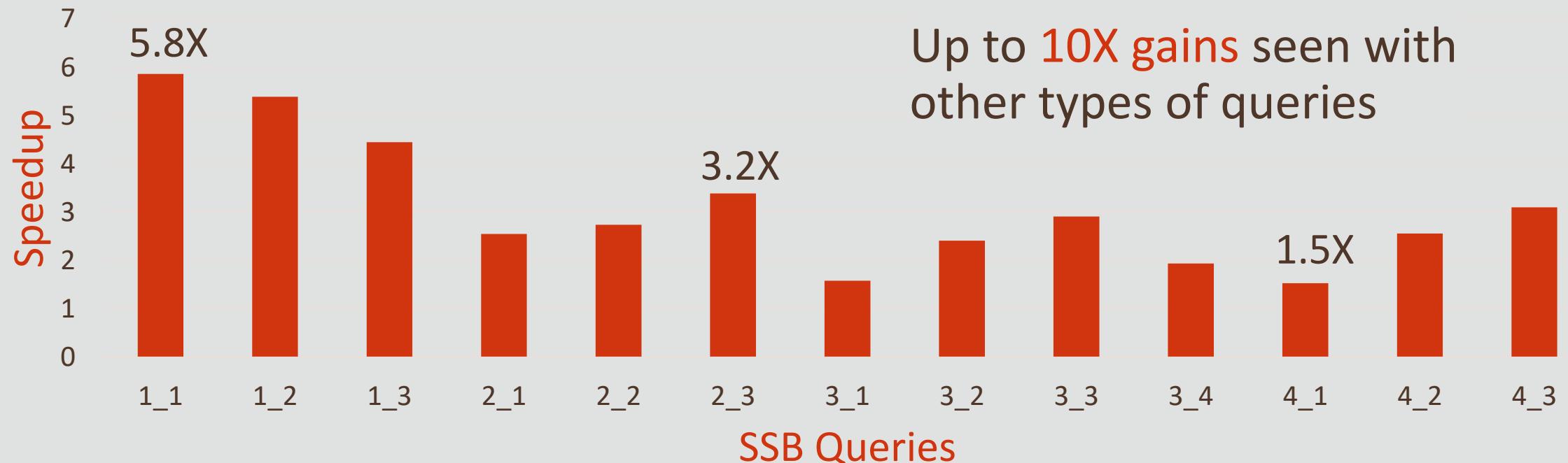
In-Memory **Accelerates** Exadata Flash Cache

- In-Memory format in Smart Columnar Flash
 - Enables **SAME** in-memory optimizations on data in Exadata flash as on Exadata DB compute nodes DRAM
 - Extends in-memory processing to Storage
 - **15x** Columnar Capacity (**100s** of TB on full rack)
- In-memory format – offloaded queries **10x faster**
 - Huge advantage over other in-memory databases and storage arrays !!!!!
- Completely automatic -no user intervention needed
 - Powers Autonomous Database



Database In-Memory Formats in Flash Cache (*CellMemory*) : Performance

CellMemory speedup over Columnar Flash Cache

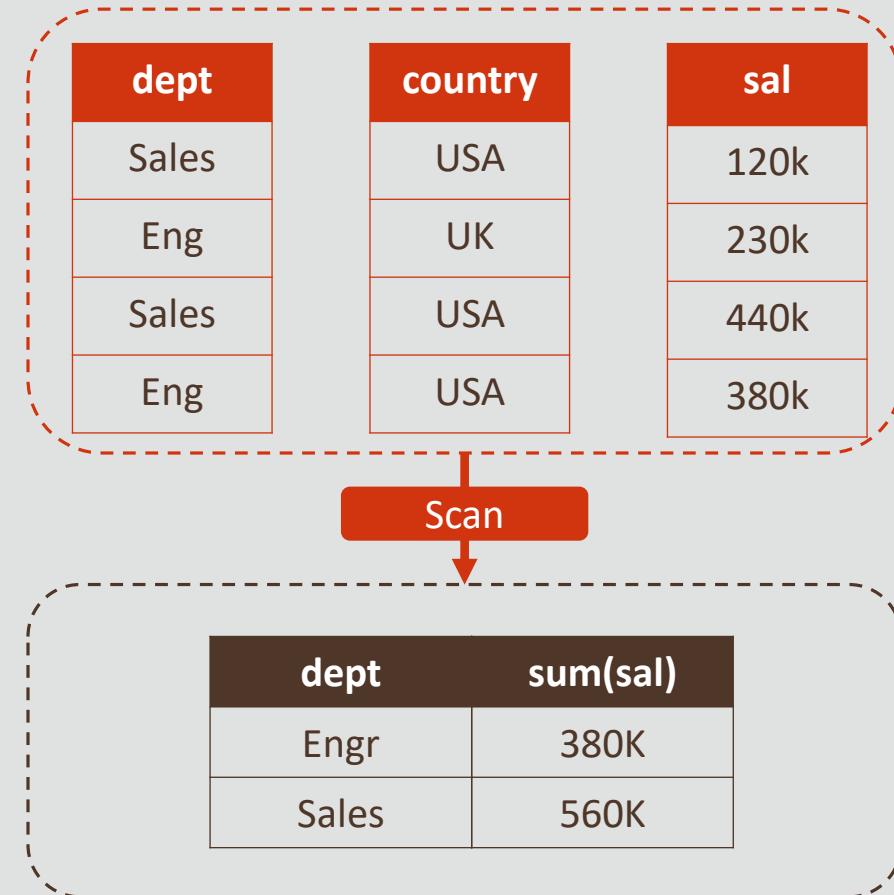


Example Benefit of In-Memory on Flash: Aggregation Offload

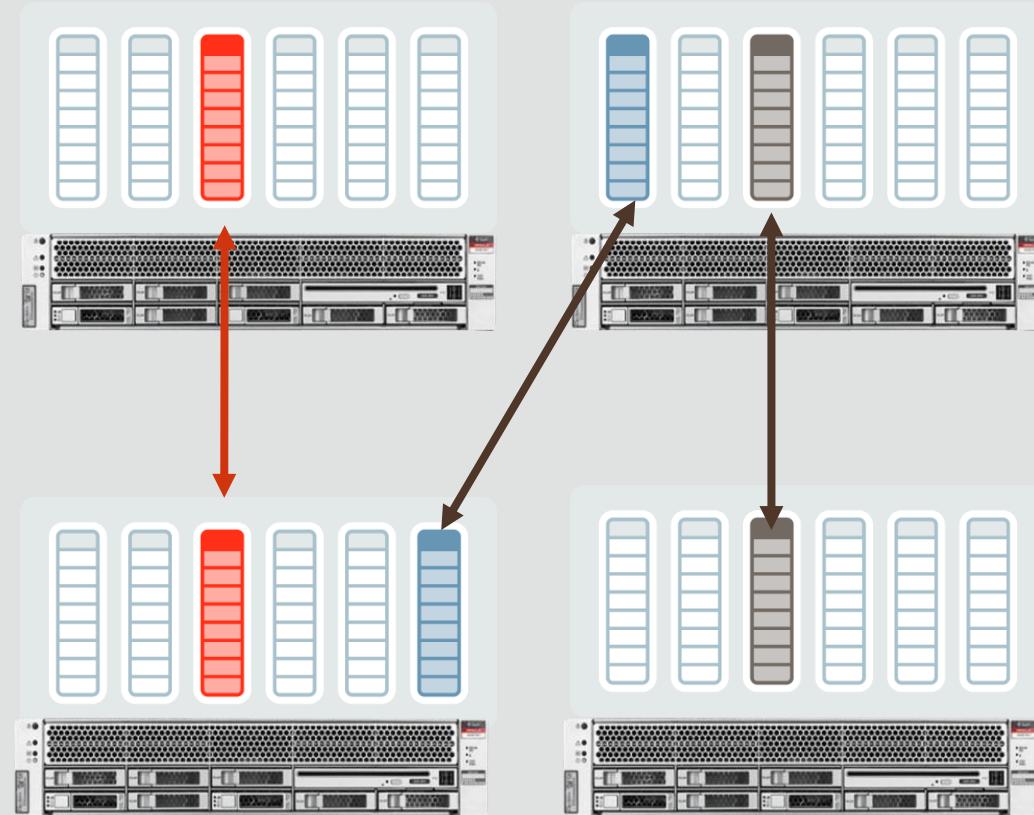
- In-Memory Format on Exadata Flash allows SUM and GROUP BY aggregations to be offloaded to storage servers:
 - Reduces data sent to the database server
 - Improves CPU utilization on the database server
- Example:

```
select dept, sum(sal) from emp  
      where country='USA' group by dept
```

 - Sum , group by operations performed on storage server
- **2x faster aggregation queries and reduced DB server**

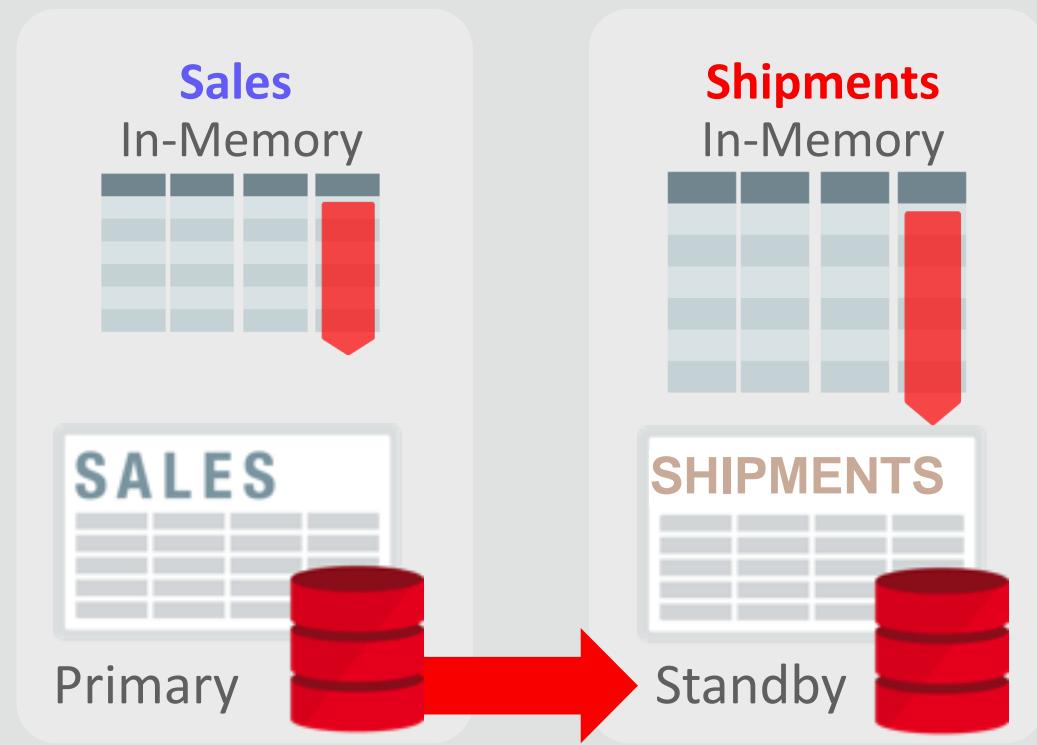


In-Memory Duplication: Fault Tolerance and Performance



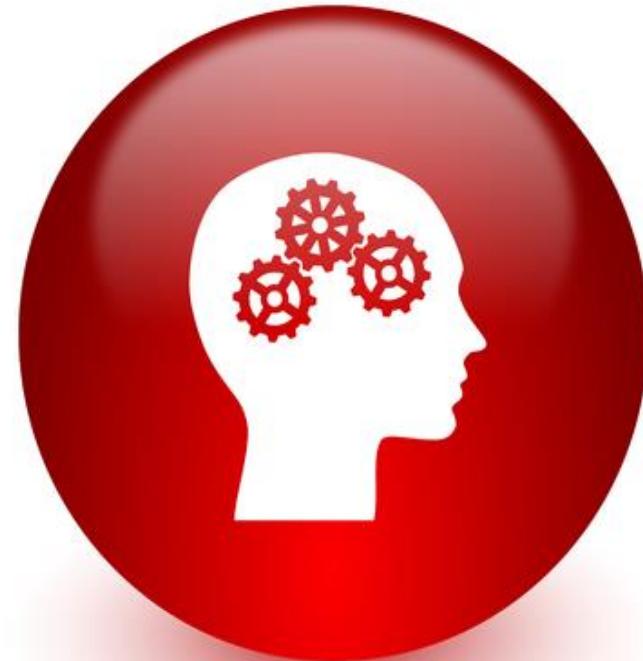
- Optionally duplicate in-memory columns across 2 nodes
 - Like storage mirroring
 - Can also duplicate across ALL nodes (e.g. small dimension tables)
 - Enabled per table/partition
 - Application transparent
- Eliminates column store repopulate after failure
- Improves performance due to greater locality

In-Memory on Active Data Guard



***Primary Database or Data Guard
Standby must be on Exadata***

- Inmemory queries can run on Active Data Guard standby
 - No impact on primary database
 - Full use of standby database resources
- Standby can have different in-memory contents from Primary
 - Increases total effective inmemory columnar capacity
 - Increases column store availability:
 - Reporting workload on standby unaffected by primary site outage

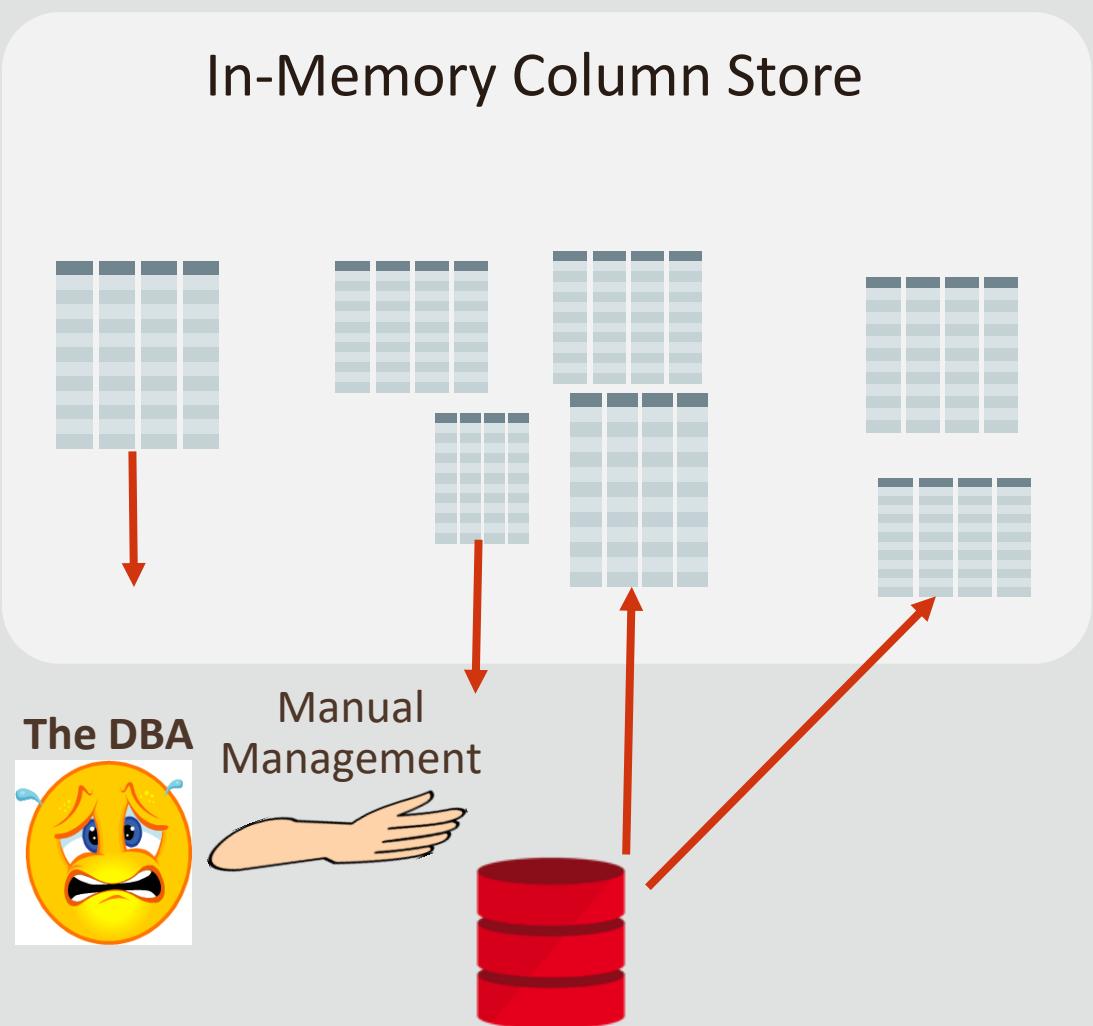


#4

Intelligent Automation

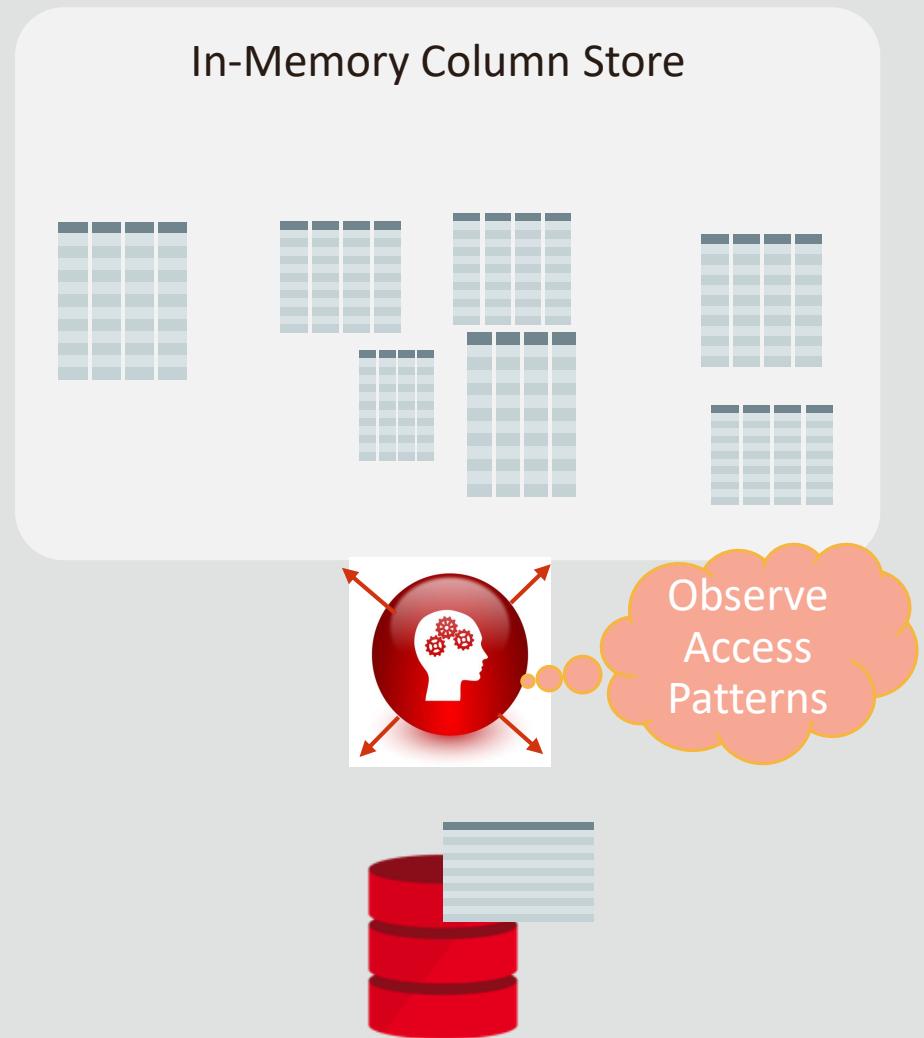
*Automatic In-Memory
Management & Storage Tiering*

Manual In-Memory Management



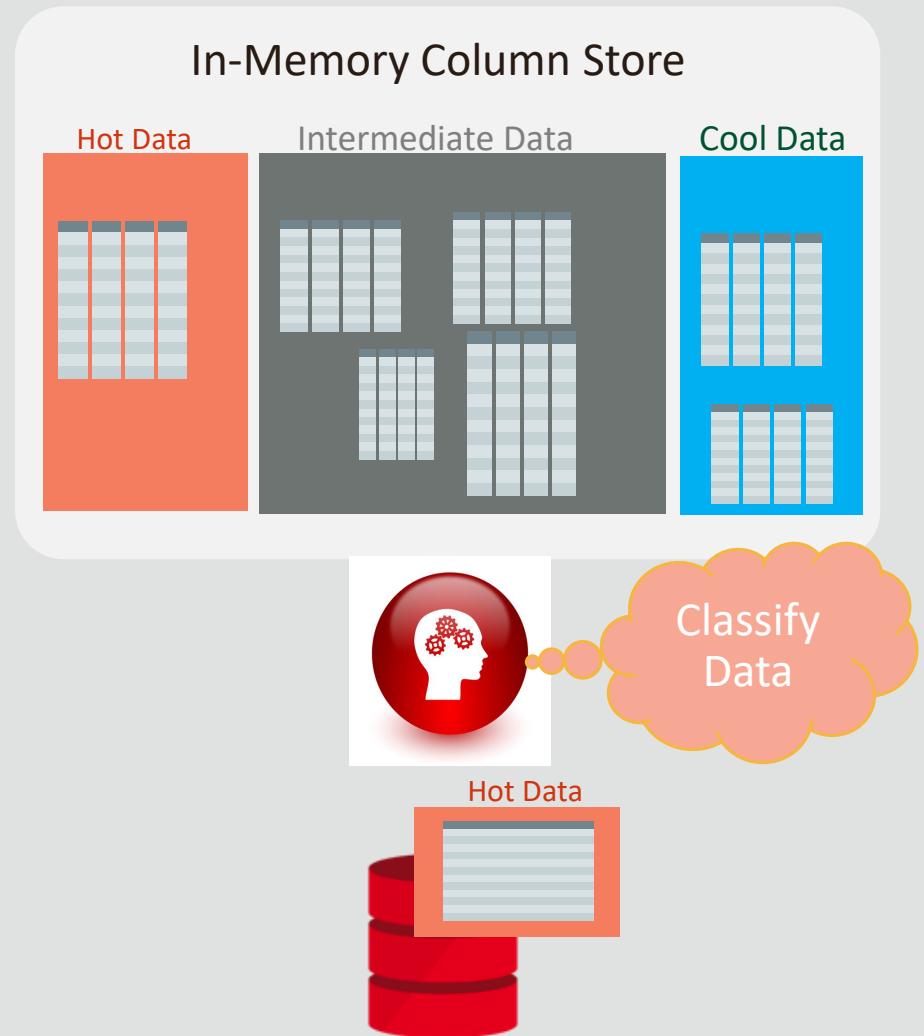
- If entire database fits within in-memory area, no need for DBA involvement!
- Otherwise, need to intelligently select in-memory candidates
- **Desired outcome:** Keep hot objects in-memory, remove colder objects
 - Access patterns are not known in advance and change over time
 - Hard for DBAs to achieve manually

Automatic In-Memory



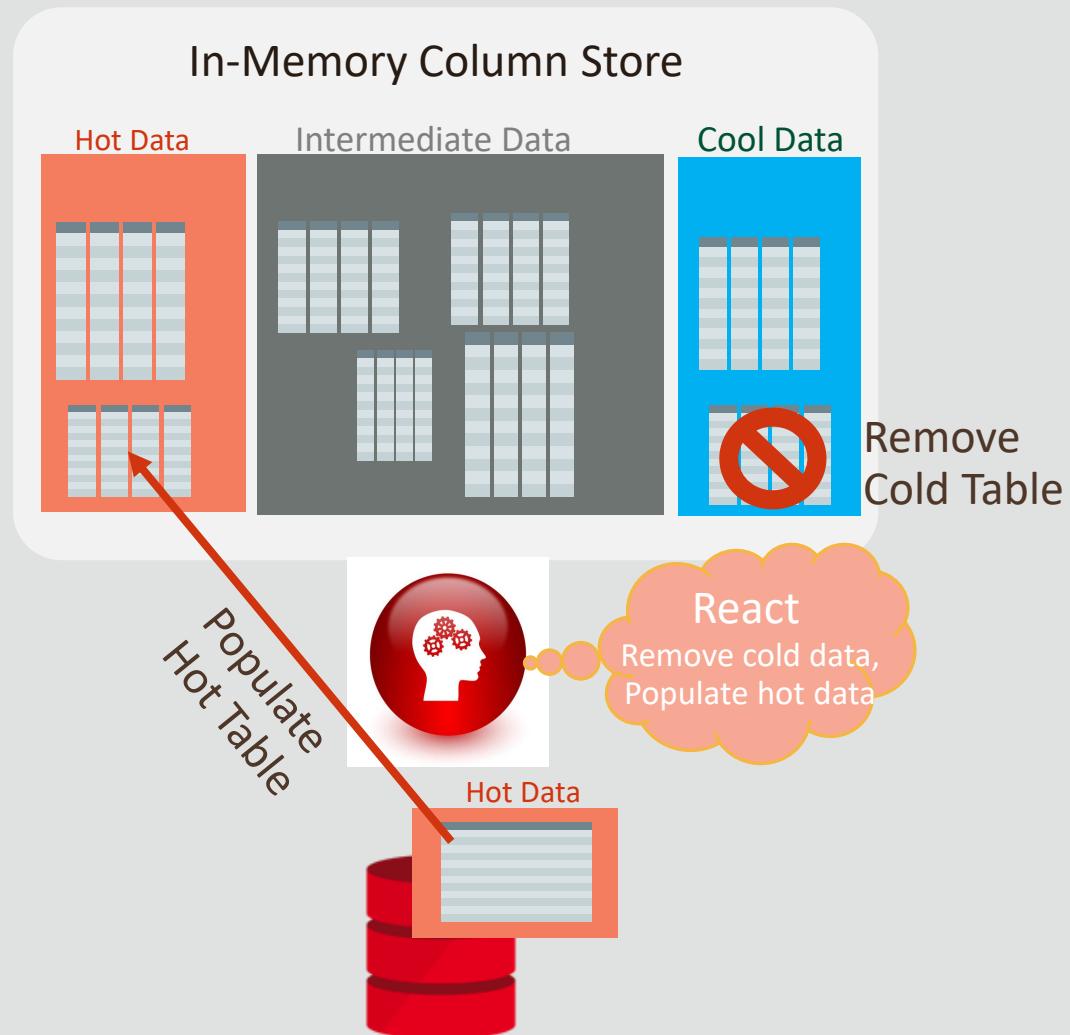
- Eliminates trial and error regarding in-memory area contents
- Constant background action:
 - Classifies data as hot, intermediate or cold
 - Hotter in-memory tables automatically populated
 - Colder in-memory tables automatically removed
 - Intelligent algorithm takes into account space-benefit tradeoffs
- Controlled by new parameter **inmemory_automatic_level**
- Useful for autonomous cloud services since no user intervention required

Automatic In-Memory



- Eliminates trial and error regarding in-memory area contents
- Constant background action:
 - Classifies data as hot, intermediate or cold
 - Hotter in-memory tables automatically populated
 - Colder in-memory tables automatically removed
 - Intelligent algorithm takes into account space-benefit tradeoffs
- Controlled by new parameter **inmemory_automatic_level**
- Useful for autonomous cloud services since no user intervention required

Automatic In-Memory



- Eliminates trial and error regarding in-memory area contents
- Constant background action:
 - Classifies data as hot, intermediate or cold
 - Hotter in-memory tables automatically populated
 - Colder in-memory tables automatically removed
 - Intelligent algorithm takes into account space-benefit tradeoffs
- Controlled by new parameter **inmemory_automatic_level**
- Useful for autonomous cloud services since no user intervention required

Preview | Hybrid In-Memory Scans

- Large, infrequently accessed columns can be excluded from the in-memory column store
 - e.g. Images, Documents, etc.
- Current behavior:** In-Memory access disallowed if query accesses any excluded column
- 20c: Hybrid In-Memory Scans**
 - Scan/filter using in-memory column store
 - Fetch excluded column values from row store
 - Over 10x performance improvement

*SELECT Invoice FROM Sales
WHERE Price > 1000*

In-Memory Column Store
SALES table
Excludes Invoice Column

ID	Item	Price
5	Camera	\$200
6	Laptop	\$2000
7	Phone	\$500
8	LED TV	\$3000

1. Scan and filter by Price using column store

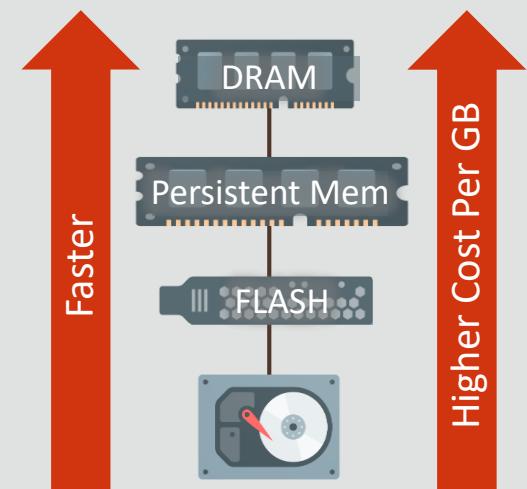
Row Store (Buffer Cache)
SALES table

ID	Item	Price	Invoice
5	Camera	\$200	
6	Laptop	\$2000	checkmark
7	Phone	\$500	
8	LED TV	\$3000	checkmark

2. Fetch invoices from row store

Preview | Persistent Memory

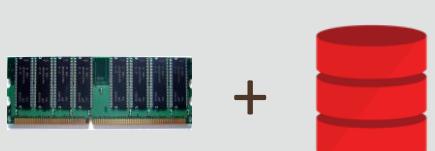
- Persistent memory is a new silicon technology
 - Capacity, performance, and price are between DRAM and flash
- Intel® Optane™ DC Persistent Memory:
 - Reads at memory speed – much faster than flash
 - Writes survive power failure unlike DRAM
- Exadata implements sophisticated algorithms to maintain integrity of data on PMEM during failures
 - Call special instructions to flush data from CPU cache to PMEM
 - Complete or backout sequence of writes interrupted by a crash



Preview | Persistent Memory (In-Memory)

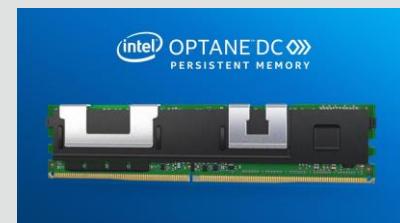
Today (Baseline)

- Not all data can fit into Memory
- Queries go against column store in DRAM and row store on DISK
- DRAM Dimms up to 128GB, and very expensive.



New : Intel® Optane™ DC Persistent Memory

- Entire workload can fit into Memory
- With Memory Mode, hottest tables are cached in DRAM for fastest access
- Apache Dimms up to 512GB

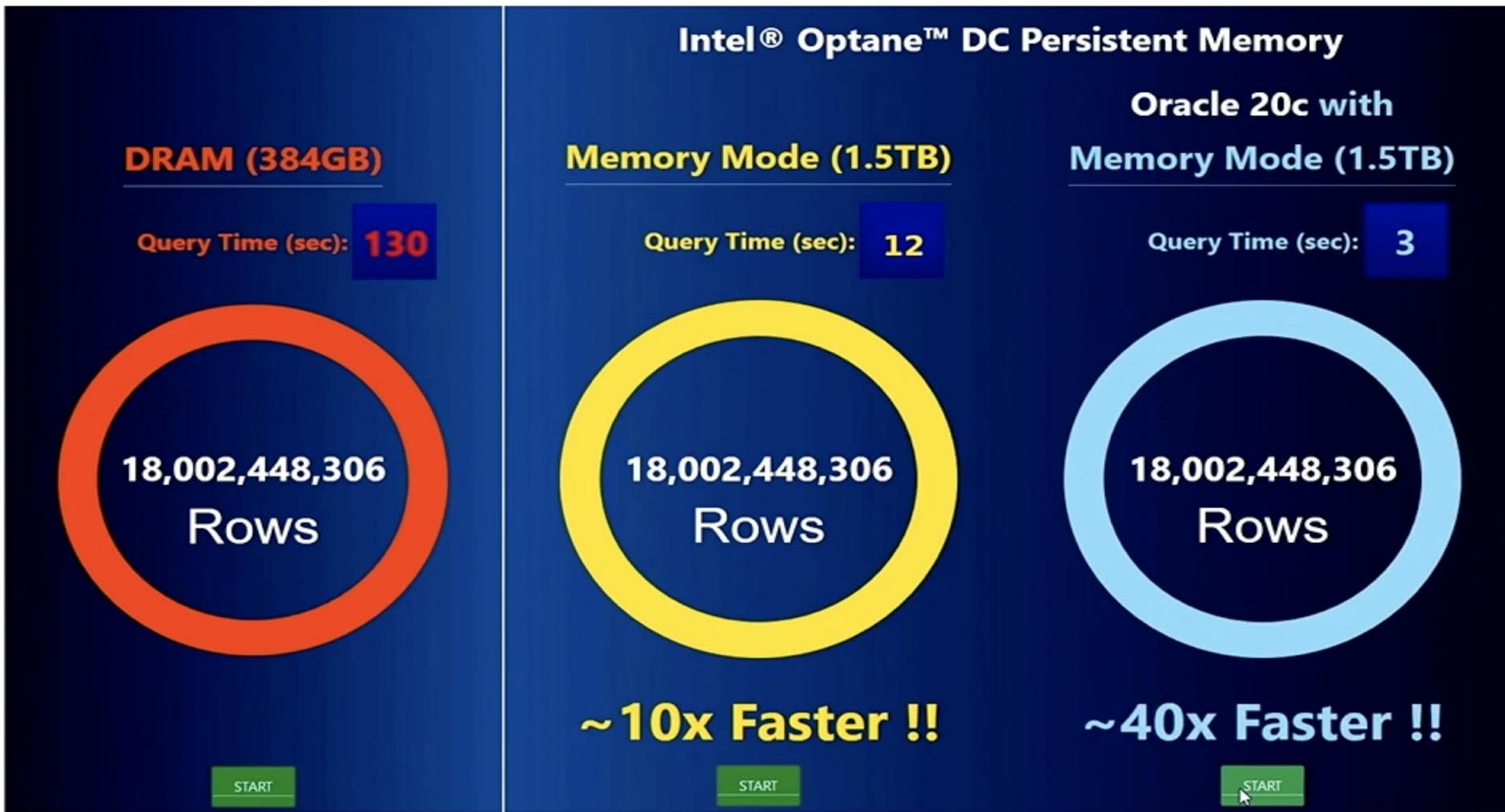


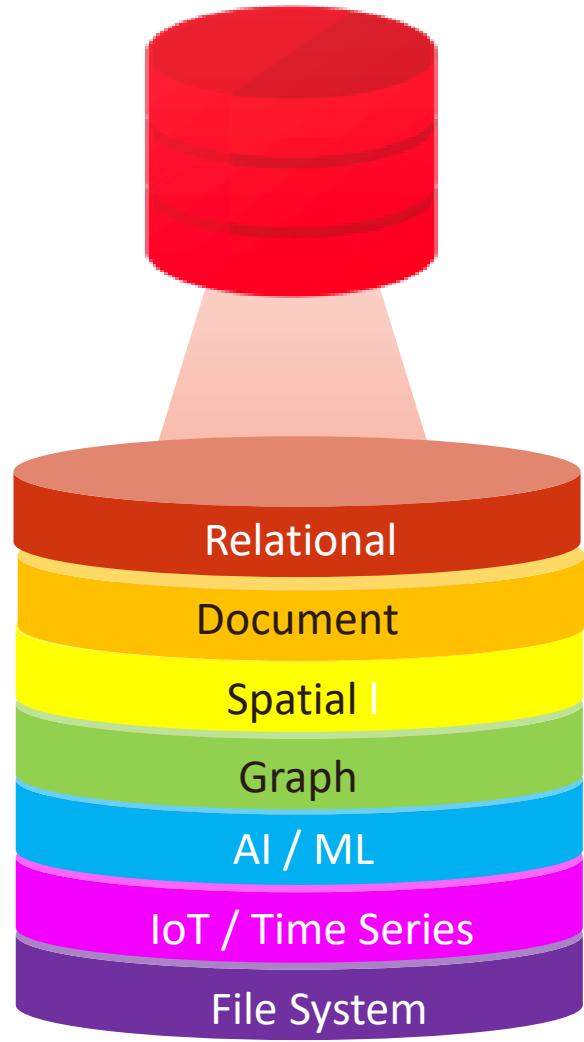
New : Intel® Optane™ DC Persistent Memory + Oracle 20c

- Oracle 20c introduces new *Deep Vectorization* framework that extends vector processing to all SQL operators



Intel® Optane™ DC Persistent Memory with Oracle In-Memory Database

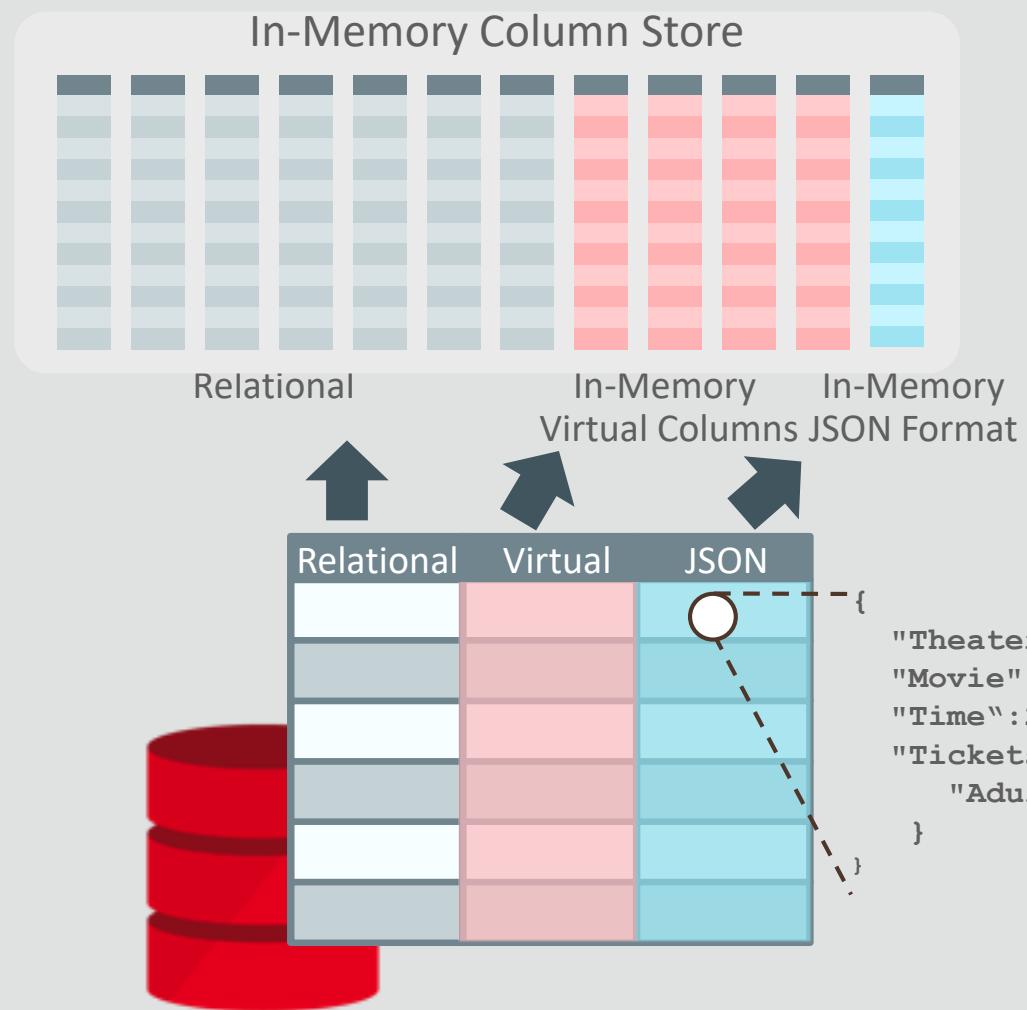




#5 Converged Analytics

*One Database for All : Relational,
Text, JSON, Spatial,...*

Faster Converged Analytics | In-Memory JSON



- Full JSON documents populated using an optimized binary format
 - Additional expressions can be created on JSON columns (e.g. JSON_VALUE) & stored in column store
 - Queries on JSON content or expressions automatically directed to In-Memory format
 - E.g. Find movies where movie.name contains “Rogue”
 - **20 - 60x** performance gains observed

In-Memory For External Tables

Fast Analytics on External Data

External Tables allow transparent SQL on external data

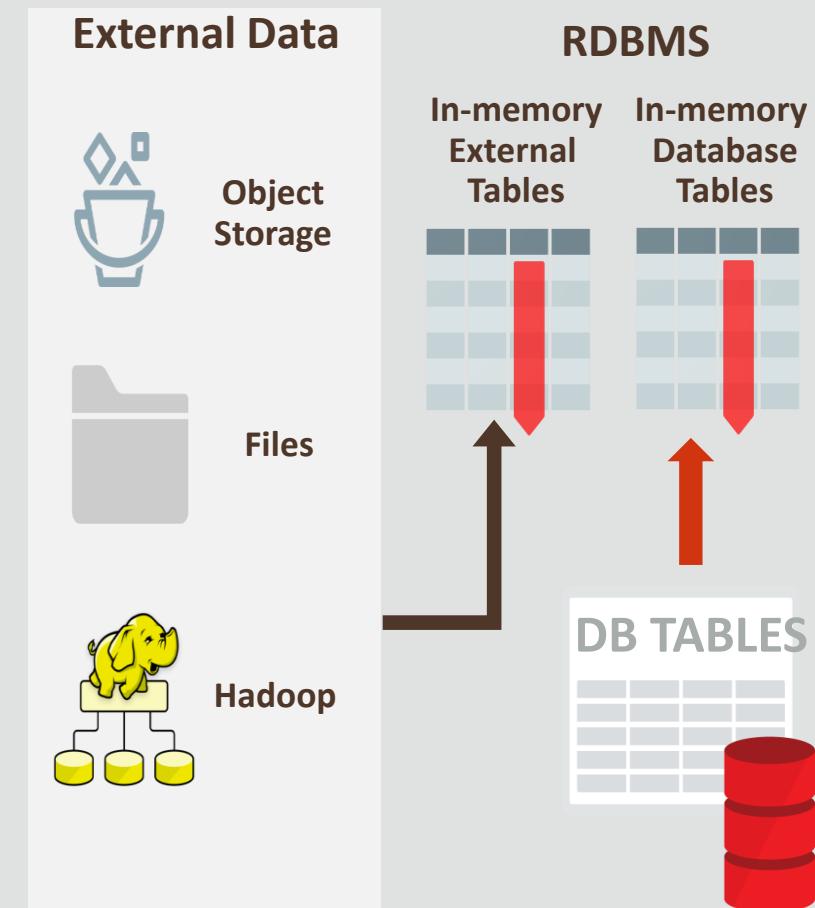
In-Memory External Tables: 100x faster analytics on external data

All In-Memory Optimizations

Vector processing, JSON expressions extend transparently to external data

Simple to enable:

```
create table EXT1 (...) organization  
external (...) inmemory
```



Preview | In-Memory Spatial Analytics

- In-Memory only *Spatial Summary* column added to each spatial column
 - Compact approximation of complex spatial detail
 - Stored in optimized In-Memory format
 - Quickly filter using SIMD vector scans
 - Replace R-Tree Indexes for Spatial Analytics
- Spatial Queries up to **10x** faster
 - No analytic R-tree index maintenance needed

In-Memory (IM) Table Columns			Additional IM columns
Parcel Number	Parcel Address	Spatial Details	Spatial Summary
095040390	300 Oracle Pkwy		
095040310	400 Oracle Pkwy		
095040250	500 Oracle Pkwy		
095040260	600 Oracle Pkwy		

Which parcel is the utility valve located in?

Search 140 Million US land parcels

Preview | In-Memory Text Analytics

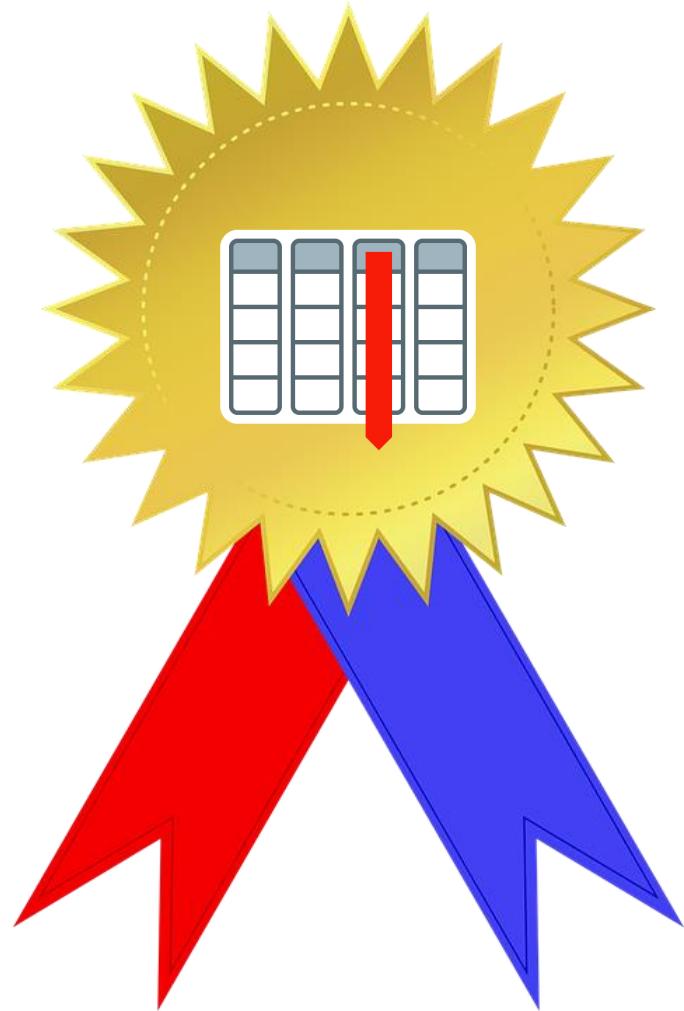
**NEW IN
20^c**

*Find job candidates with "PhD" degrees
who have "database" in their resumes*

In-Memory Column Store

Name	Degree	Resume (Text)	Text Index
John	PhD	<p>INTRODUCTION</p> <p>I'm John, I suggest that it's not an overstatement to say that reading resumes is my job. Most of the time I spend about 10 minutes on each resume. I've never seen a resume that took more than 10 minutes to read. The specific character you're looking for may be very specific or very general. I can either define it in a few words or I can define it in a few sentences.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p>	<p>Words</p> <p>..</p> <p>..</p>
Ram	BS	<p>INTRODUCTION</p> <p>I'm Ram, I suggest that it's not an overstatement to say that reading resumes is my job. Most of the time I spend about 10 minutes on each resume. I've never seen a resume that took more than 10 minutes to read. The specific character you're looking for may be very specific or very general. I can either define it in a few words or I can define it in a few sentences.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p>	<p>..</p> <p>..</p> <p>..</p>
Emily	MS	<p>INTRODUCTION</p> <p>I'm Emily, I suggest that it's not an overstatement to say that reading resumes is my job. Most of the time I spend about 10 minutes on each resume. I've never seen a resume that took more than 10 minutes to read. The specific character you're looking for may be very specific or very general. I can either define it in a few words or I can define it in a few sentences.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p>	<p>database</p> <p>..</p> <p>..</p>
Sara	MS	<p>INTRODUCTION</p> <p>I'm Sara, I suggest that it's not an overstatement to say that reading resumes is my job. Most of the time I spend about 10 minutes on each resume. I've never seen a resume that took more than 10 minutes to read. The specific character you're looking for may be very specific or very general. I can either define it in a few words or I can define it in a few sentences.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p> <p>For example, if I am in one of three situations, this will tell me what kind of resume I am looking at. If I am looking for a job, I am looking for a specific character. If I am looking for a friend, I am looking for a general character. If I am looking for a partner, I am looking for a partner character.</p>	<p>..</p> <p>..</p> <p>..</p>

- In-Memory only *Inverted Index* added to each text column
 - Maps words to documents which *contain* those words
 - Replaces on-disk text index for analytic workloads
 - Converged queries (*relational + text*) can benefit from in-memory
 - 3x faster

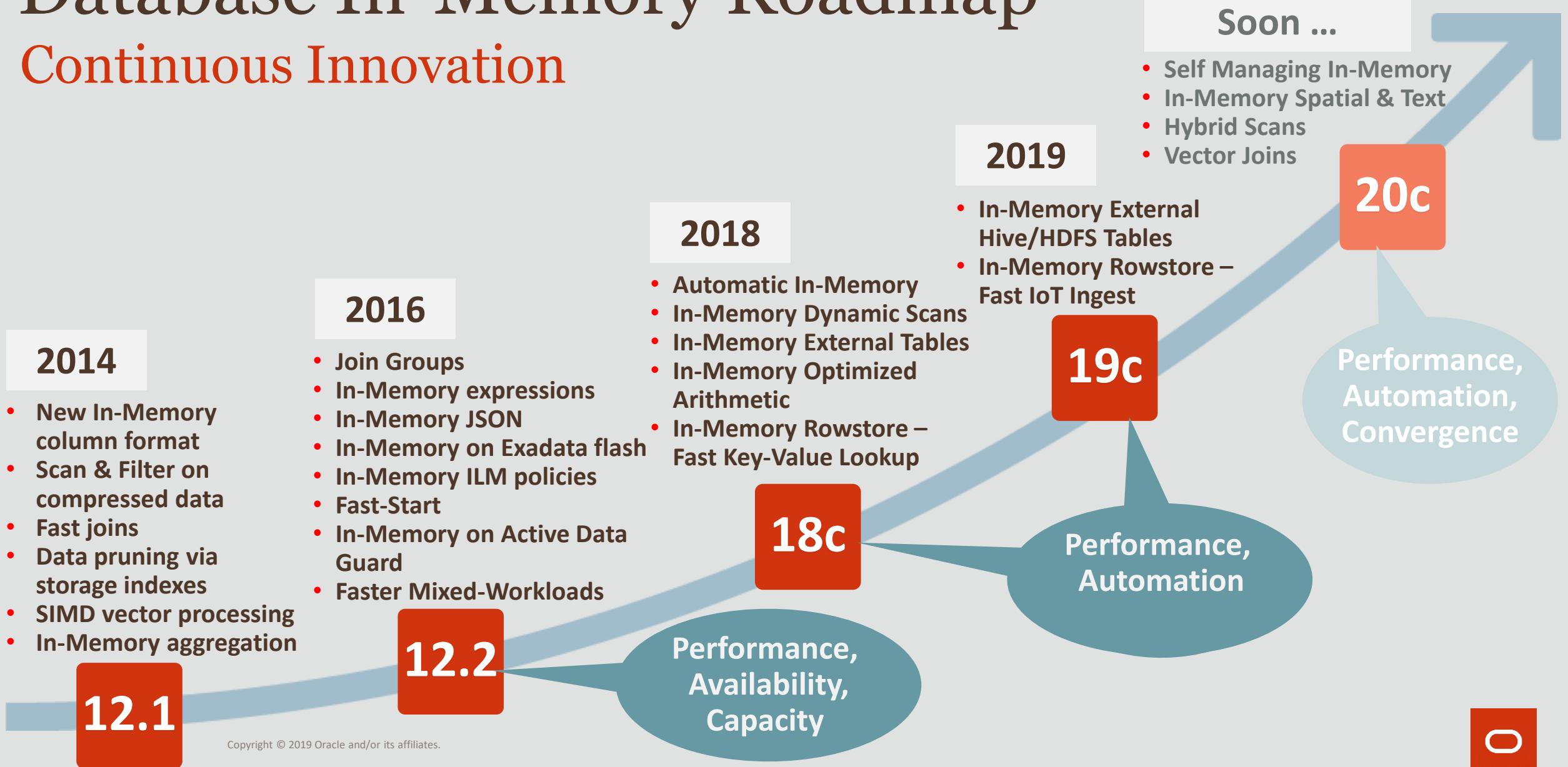


Top-5 Innovations Summary



Database In-Memory Roadmap

Continuous Innovation

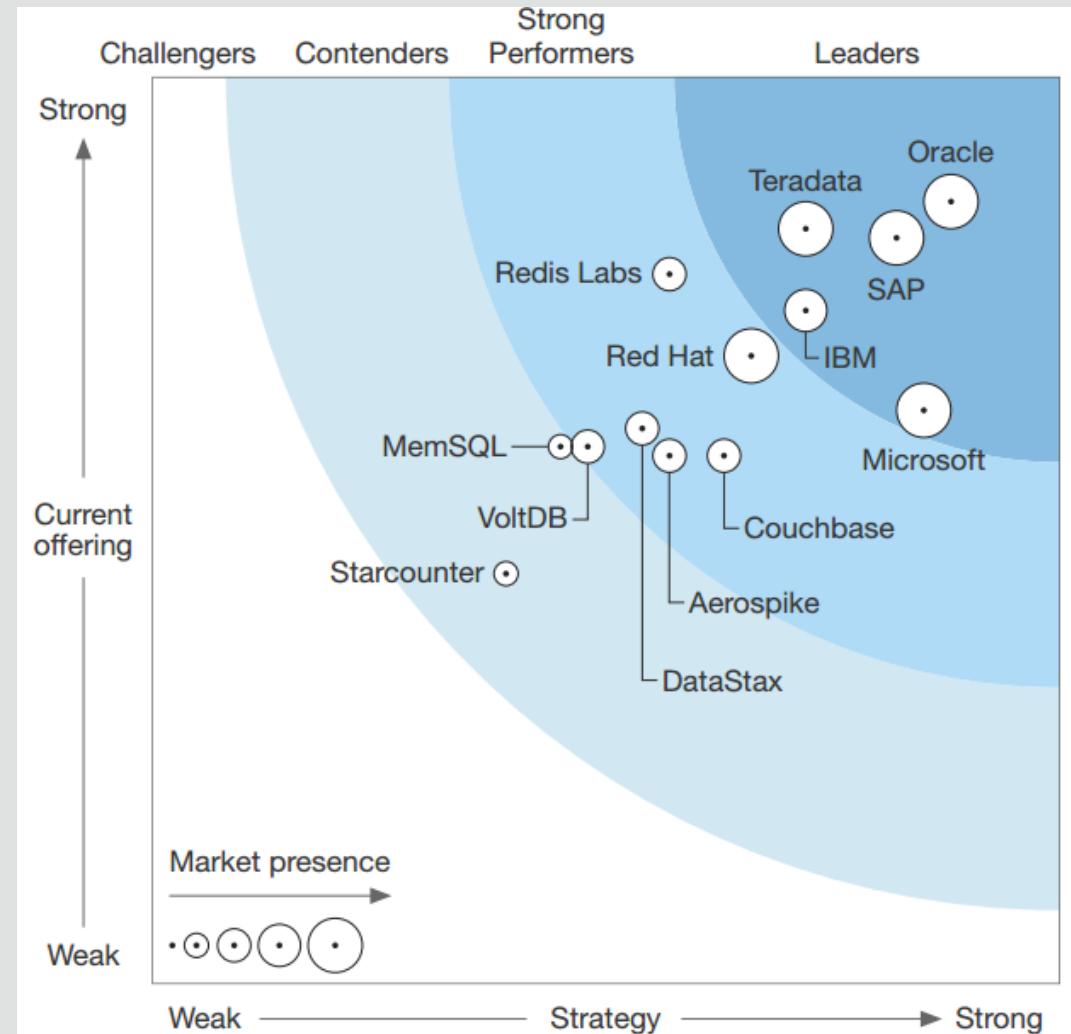


The Forrester Wave™: In-Memory Databases, Q1 2017

**Oracle In-Memory Databases
Scored Highest by Forrester
on both Current Offering
and Strategy**

<http://www.oracle.com/us/corporate/analystreports/forrester-imdb-wave-2017-3616348.pdf>

The Forrester Wave™ is copyrighted by Forrester Research, Inc. Forrester and Forrester Wave™ are trademarks of Forrester Research, Inc. The Forrester Wave™ is a graphical representation of Forrester's call on a market and is plotted using a detailed spreadsheet with exposed scores, weightings, and comments. Forrester does not endorse any vendor, product, or service depicted in the Forrester Wave. Information is based on best available resources. Opinions reflect judgment at the time and are subject to change.



Hot off the Press: 2019 Forrester Wave Translytical Data Platforms

Oracle Position: Leader

Oracle ranked highest on both Axis

- “Unlike other vendors, Oracle uses a **dual-format database** (row and columns for the same table) to deliver optimal translytical performance.”
- “Customers like Oracle’s capability to support many workloads including OLTP, IoT, microservices, **multimodel**, data science, AI/ML, spatial, graph, and analytics”
- “Existing Oracle applications **do not require any changes** to the application in order to leverage Oracle Database In-Memory”



Additional Resources

Related White Papers

- [Oracle Database In-Memory White Paper](#)
- [Oracle Database In-Memory Aggregation Paper](#)
- [When to use Oracle Database In-Memory](#)
- [Oracle Database In-Memory Advisor](#)
- [SQL Plan Management White Paper](#)
- [POC / Implementation Guidelines](#)

Related Videos

- [In-Memory YouTube Channel](#)
- [Managing Oracle Database In-Memory](#)
- [Database In-Memory and Oracle Multitenant](#)
- [Industry Experts Discuss Oracle Database In-Memory](#)
- [Software on Silicon](#)



Additional Details

- [Oracle Database In-Memory Blog](#)
- [Optimizer blog](#)

Join the Conversation

- [@dbinmemory](#)
- <https://blogs.oracle.com/in-memory/>
- <https://www.facebook.com/OracleDatabase>
- <http://www.oracle.com/goto/dbim.html>



ORACLE

26

Final Review + Systems Potpourri



Intro to Database Systems
15-445/15-645
Fall 2019

AP

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #4: Tuesday Dec 10th @ 11:59pm

Extra Credit: Tuesday Dec 10th @ 11:59pm

Final Exam: Monday Dec 9th @ 5:30pm



FINAL EXAM

Who: You

What: <http://cmudb.io/f19-final>

When: Monday Dec 9th @ 5:30pm

Where: Porter Hall 100

Why: https://youtu.be/6yOH_FjeSAQ



FINAL EXAM

What to bring:

- CMU ID
- One page of handwritten notes (double-sided)
- Extra Credit Coupon

Optional:

- Spare change of clothes
- Food

What not to bring:

- Your roommate



COURSE EVALS

Your feedback is strongly needed:

→ <https://cmu.smartevals.com>

Things that we want feedback on:

- Homework Assignments
- Projects
- Reading Materials
- Lectures



OFFICE HOURS

Andy's hours:

- Friday Dec 6th @ 3:30-4:30pm
- Monday Dec 9th @ 1:30-2:30pm

All TAs will have their regular office hours up to and including Saturday Dec 14th



STUFF BEFORE MID-TERM

SQL

Buffer Pool Management

Hash Tables

B+ Trees

Storage Models

Inter-Query Parallelism



TRANSACTIONS

ACID

Conflict Serializability:

- How to check?
- How to ensure?

View Serializability

Recoverable Schedules

Isolation Levels / Anomalies



TRANSACTIONS

Two-Phase Locking

- Rigorous vs. Non-Rigorous
- Deadlock Detection & Prevention

Multiple Granularity Locking

- Intention Locks



TRANSACTIONS

Timestamp Ordering Concurrency Control

→ Thomas Write Rule

Optimistic Concurrency Control

→ Read Phase

→ Validation Phase

→ Write Phase

Multi-Version Concurrency Control

→ Version Storage / Ordering

→ Garbage Collection



CRASH RECOVERY

Buffer Pool Policies:

- STEAL vs. NO-STEAL
- FORCE vs. NO-FORCE

Write-Ahead Logging

Logging Schemes

Checkpoints

ARIES Recovery

- Log Sequence Numbers
- CLRs



DISTRIBUTED DATABASES

System Architectures

Replication

Partitioning Schemes

Two-Phase Commit



2018

	Cockroach LABS	26
	Spanner	25
	mongoDB	24
	Aurora	18
	redis	18
	cassandra	17
	elasticsearch	12
	HIVE	11
	Scuba	10
	MySQL™	10

2019

	Scuba	20
	mongoDB	19
	Cockroach LABS	18
	Aurora	17
	Spanner	17
	snowflake	17
	PostgreSQL	17
	OceanBase	16
	amazon REDSHIFT	15
	elasticsearch	15



Scuba

facebook®

FACEBOOK SCUBA

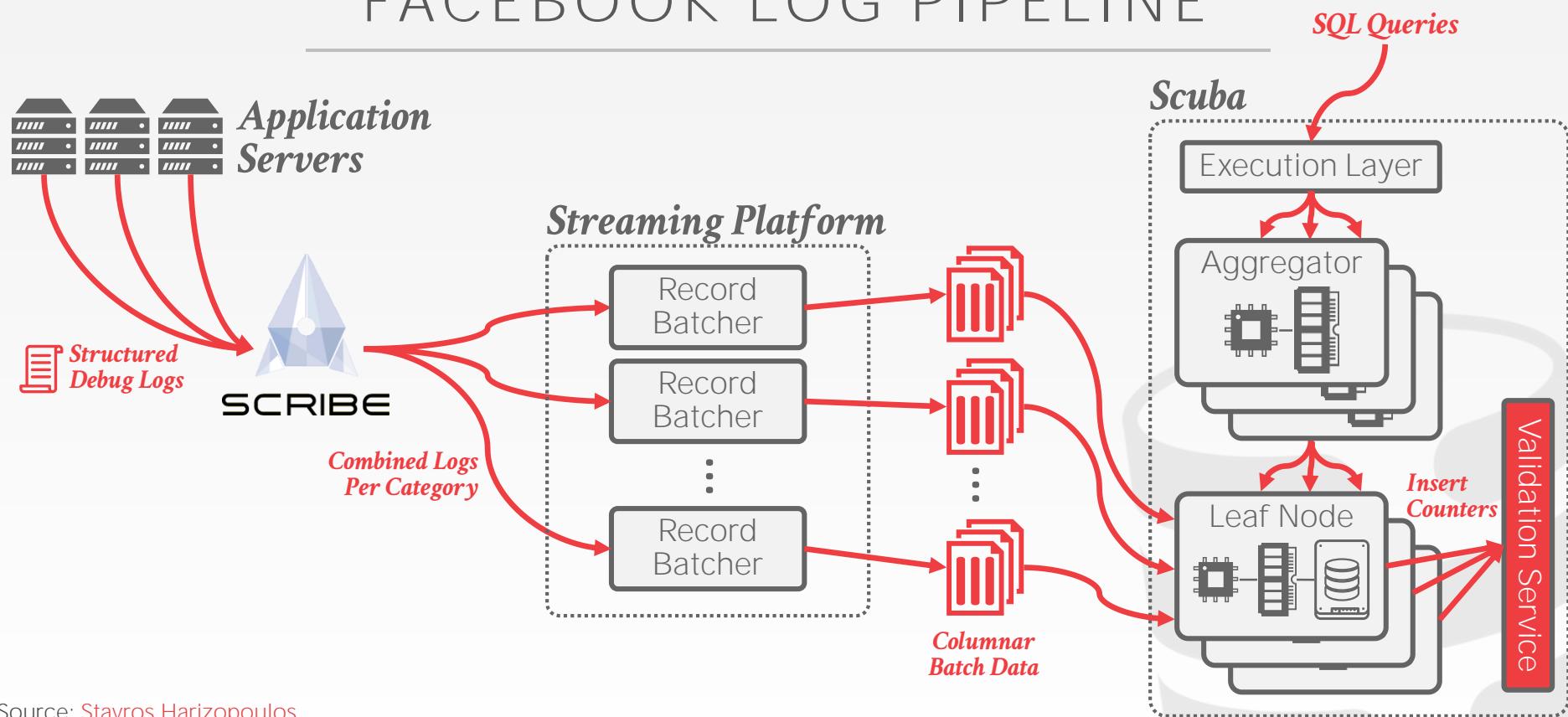
Internal DBMS designed for real-time data analysis of performance monitoring data.

- Columnar Storage Model
- Distributed / Shared-Nothing
- Tiered-Storage
- No Joins or Global Sorting
- Heterogeneous Hierarchical Distributed Architecture

Designed for low-latency ingestion and queries.

Redundant deployments with lossy fault-tolerance.

FACEBOOK LOG PIPELINE



Source: Stavros Harizopoulos

SCUBA ARCHITECTURE

Leaf Nodes:

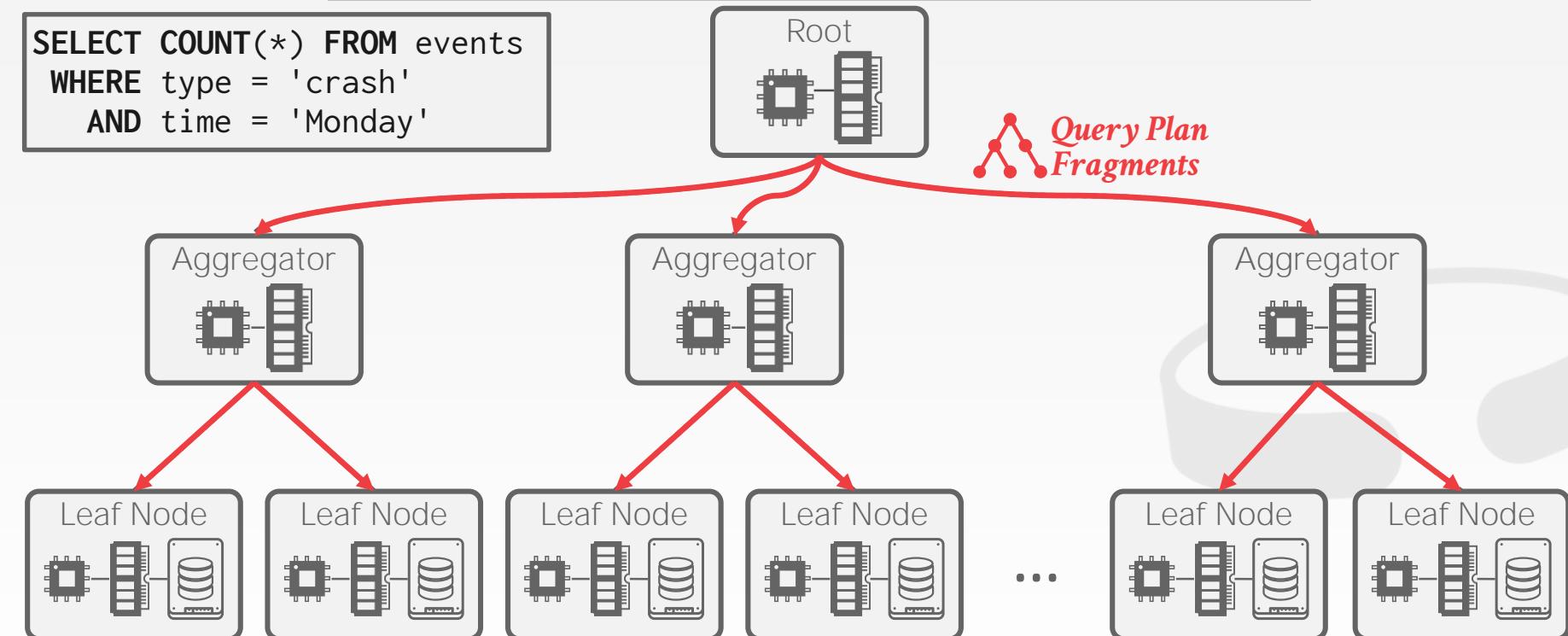
- Store columnar data on local SSDs.
- Leaf nodes may or may not contain data needed for a query.
- No indexes. All scanning is done on time ranges.

Aggregator Nodes:

- Dispatch plan fragments to all its children leaf nodes.
- Combine the results from children.
- If a leaf node does not produce results before a timeout, then they are omitted.

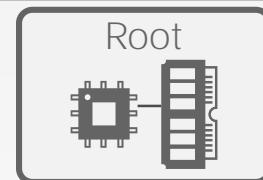
SCUBA ARCHITECTURE

```
SELECT COUNT(*) FROM events  
WHERE type = 'crash'  
AND time = 'Monday'
```

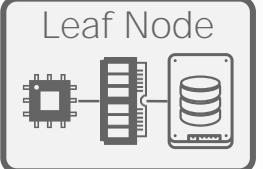
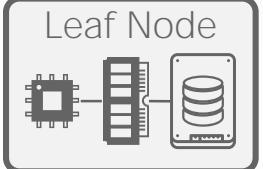
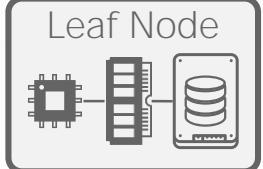
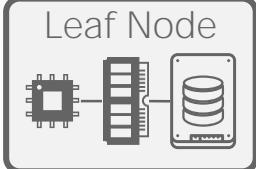
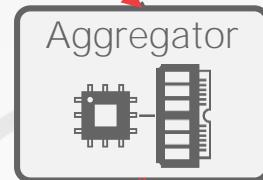
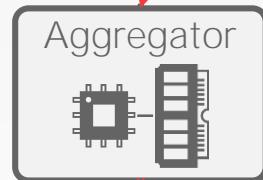
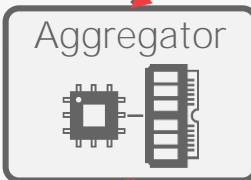


SCUBA ARCHITECTURE

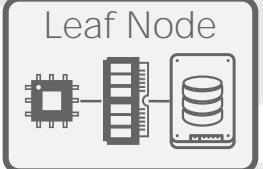
```
SELECT COUNT(*) FROM events  
WHERE type = 'crash'  
AND time = 'Monday'
```



Query Plan Fragments

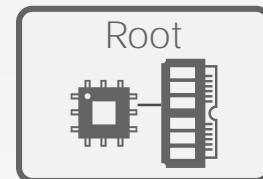


...

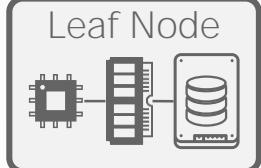
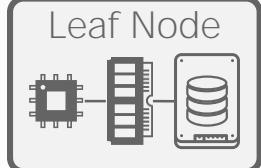
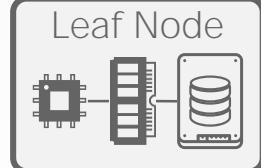
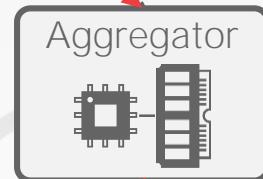
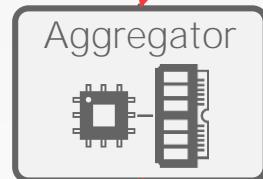
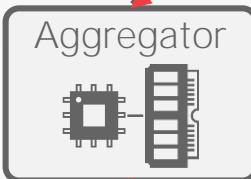


SCUBA ARCHITECTURE

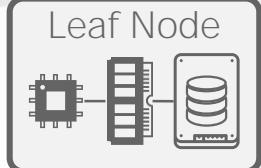
```
SELECT COUNT(*) FROM events  
WHERE type = 'crash'  
AND time = 'Monday'
```



Query Plan Fragments



...



10

20

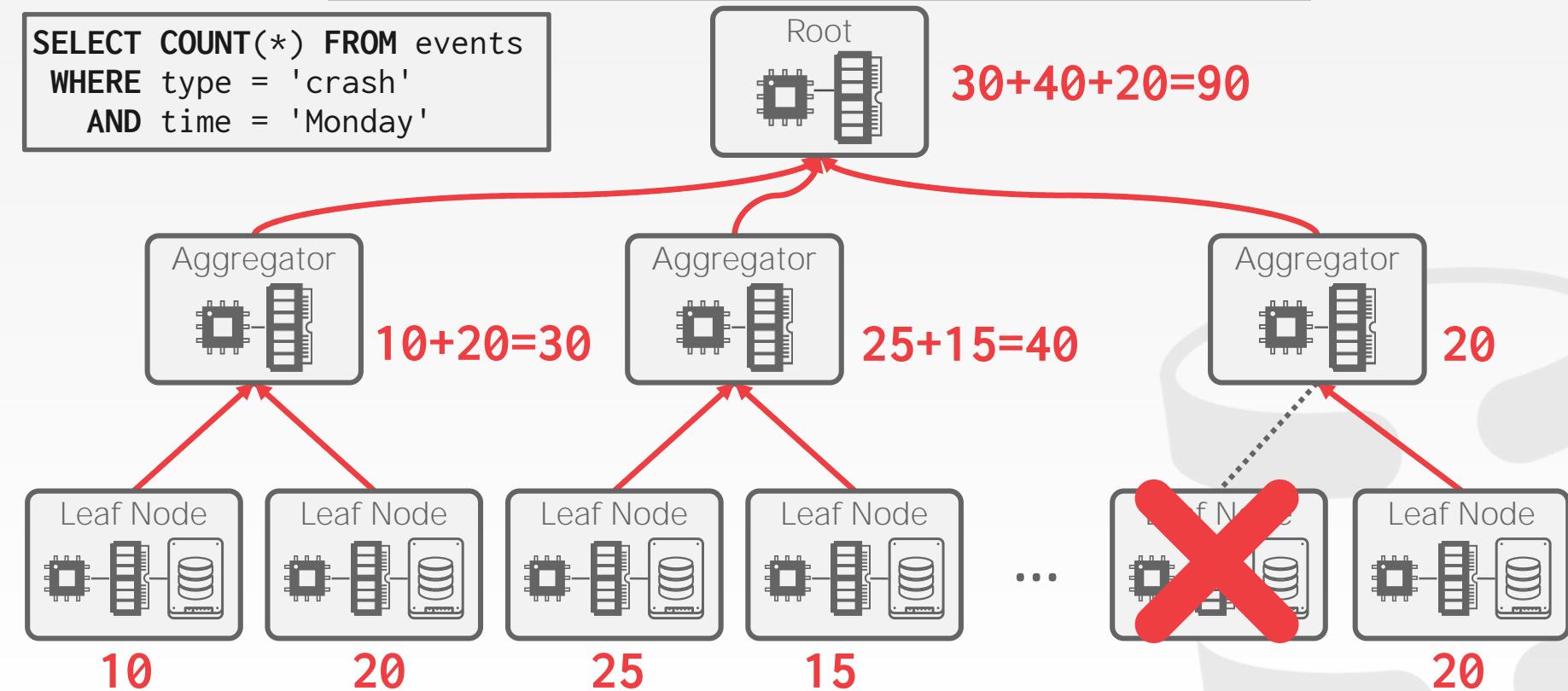
25

15

20

SCUBA ARCHITECTURE

```
SELECT COUNT(*) FROM events  
WHERE type = 'crash'  
AND time = 'Monday'
```



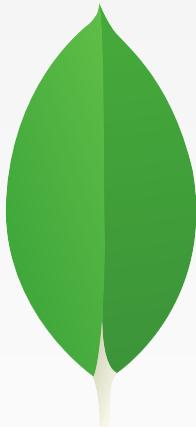
FAULT TOLERANCE

Facebook maintains multiple Scuba clusters that contain the same databases.

Every query is executed on all the clusters at the same time.

It compares the amount of missing data each cluster had when executing the query to determine which one produced the most accurate result.

→ Track the number of tuples examined vs. number of tuples inserted via Validation Service.



mongoDB

MONGODB

Distributed document DBMS started in 2007.

- Document → Tuple
- Collection → Table/Relation

Open-source (Server Side Public License)

Centralized shared-nothing architecture.

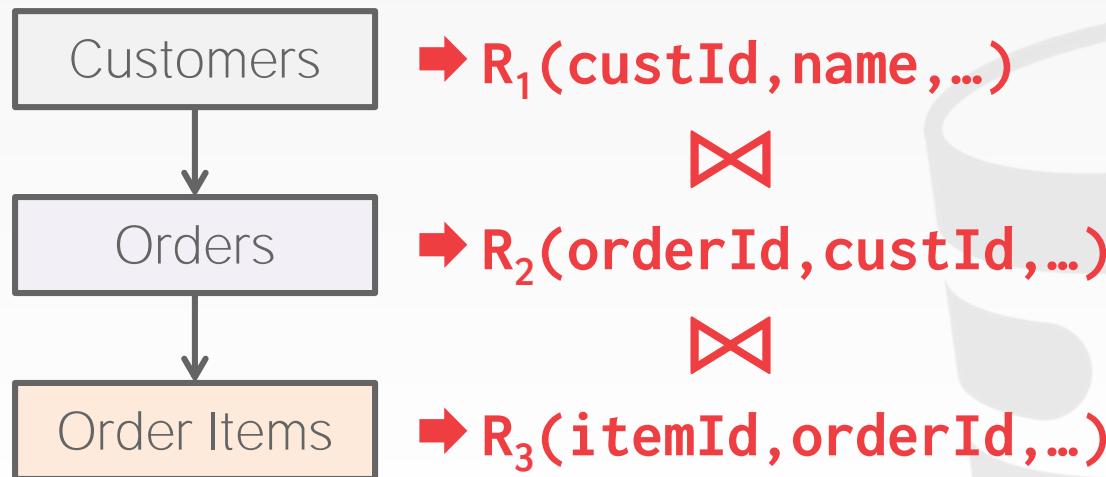
Concurrency Control:

- OCC with multi-granular locking



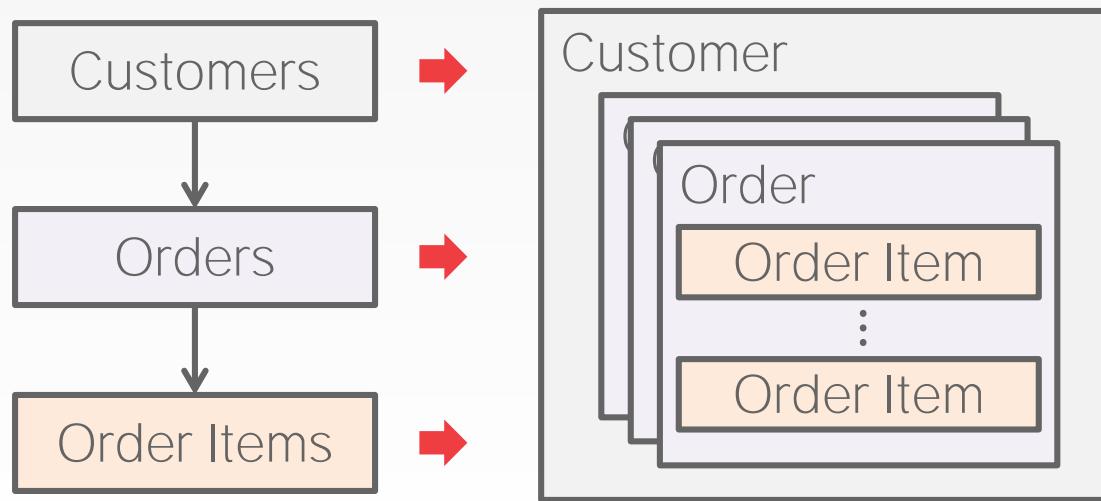
PHYSICAL DENORMALIZATION

A customer has orders and each order has order items.



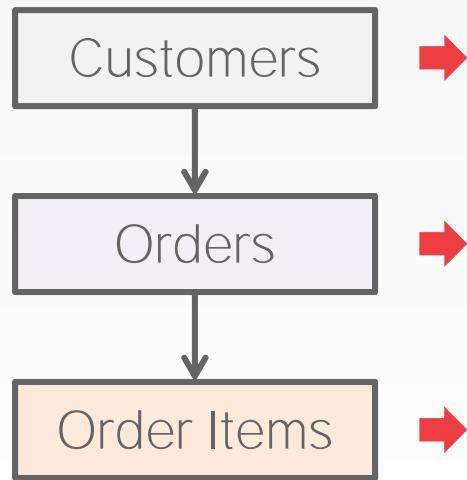
PHYSICAL DENORMALIZATION

A customer has orders and each order has order items.



PHYSICAL DENORMALIZATION

A customer has orders and each order has order items.



```
{  
  "custId": 1234,  
  "custName": "Andy",  
  "orders": [  
    { "orderId": 9999,  
      "orderItems": [  
        { "itemId": "XXXX",  
          "price": 19.99 },  
        { "itemId": "YYYY",  
          "price": 29.99 },  
      ] }  
  ]}
```

QUERY EXECUTION

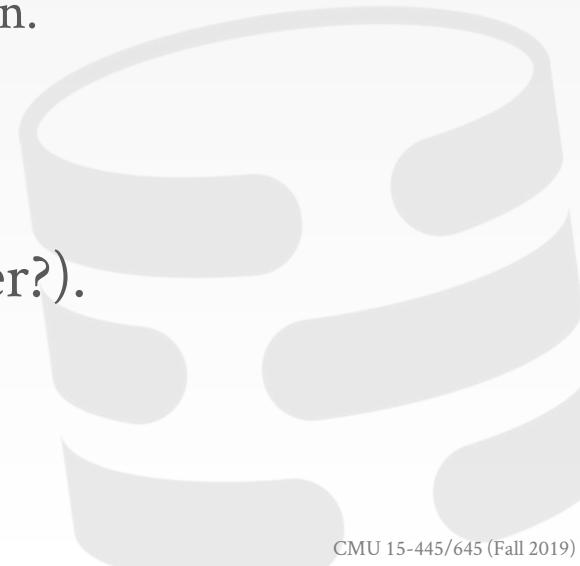
JSON-only query API

No cost-based query planner / optimizer.
→ Heuristic-based + "random walk" optimization.

JavaScript UDFs (not encouraged).

Supports server-side joins (only left-outer?).

Multi-document transactions.



DISTRIBUTED ARCHITECTURE

Heterogeneous distributed components.

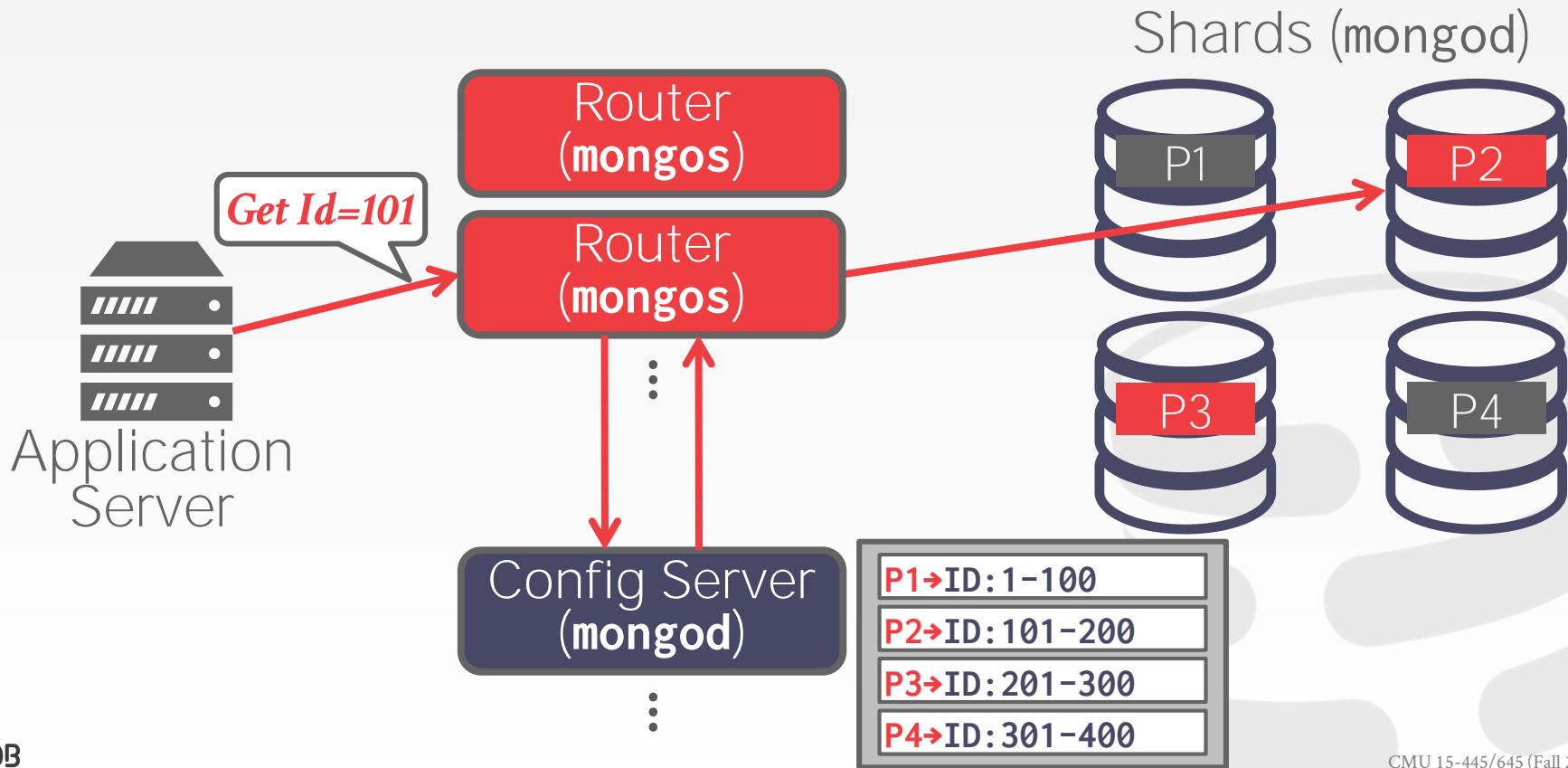
- Shared nothing architecture
- Centralized query router.

Master-slave replication.

Auto-sharding:

- Define 'partitioning' attributes for each collection (hash or range).
- When a shard gets too big, the DBMS automatically splits the shard and rebalances.

MONGODB CLUSTER ARCHITECTURE



STORAGE ARCHITECTURE

Originally used **mmap** storage manager

- No buffer pool.
- Let the OS decide when to flush pages.
- Single lock per database.

MongoDB v3 supports pluggable storage backends

- **WiredTiger** from BerkeleyDB alumni.
<http://cmudb.io/lectures2015-wiredtiger>
- **RocksDB** from Facebook (“MongoRocks”)
<http://cmudb.io/lectures2015-rocksdb>





Cockroach LABS





COCKROACHDB

Started in 2015 by ex-Google employees.

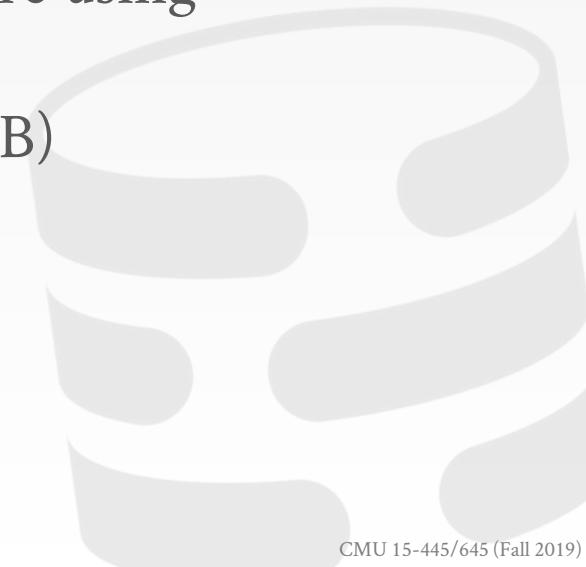
Open-source (BSL – MariaDB)

Decentralized shared-nothing architecture using range partitioning.

Log-structured on-disk storage (RocksDB)

Concurrency Control:

- MVCC + OCC
- Serializable isolation only





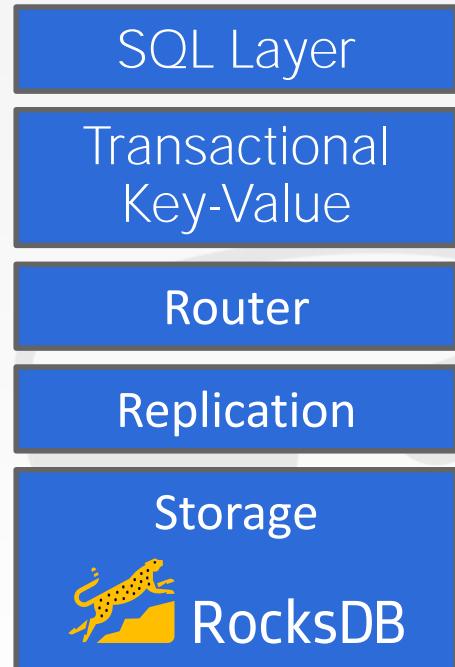
DISTRIBUTED ARCHITECTURE

Multi-layer architecture on top of a replicated key-value store.

→ All tables and indexes are stored in a giant sorted map in the k/v store.

Uses RocksDB as the storage manager at each node.

Raft protocol (variant of Paxos) for replication and consensus.



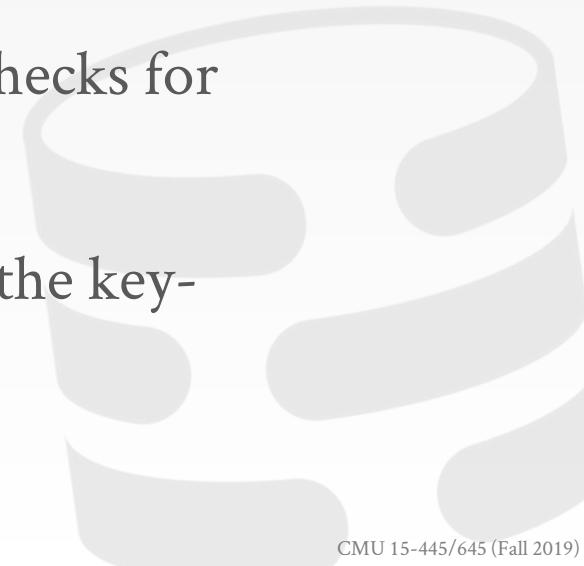


CONCURRENCY CONTROL

DBMS uses hybrid clocks (physical + logical) to order transactions globally.
→ Synchronized wall clock with local counter.

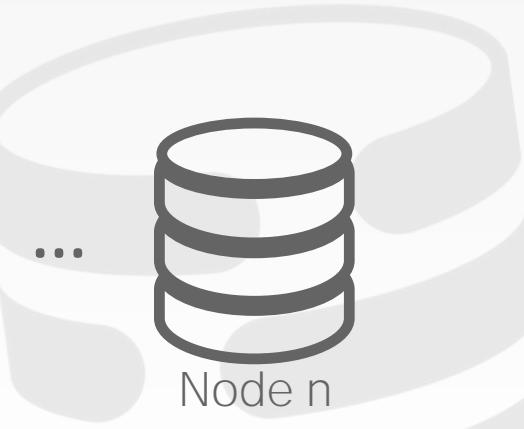
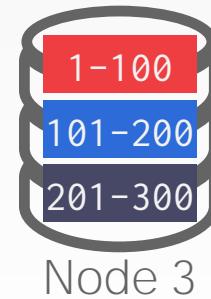
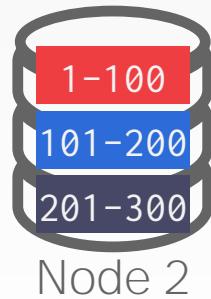
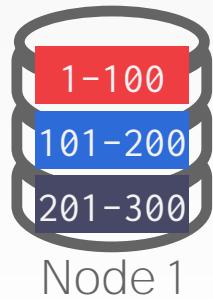
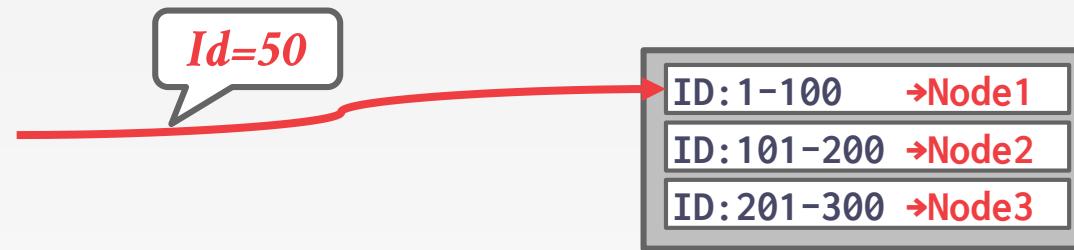
Txns stage writes as "intents" and then checks for conflicts on commit.

All meta-data about txns state resides in the key-value store.



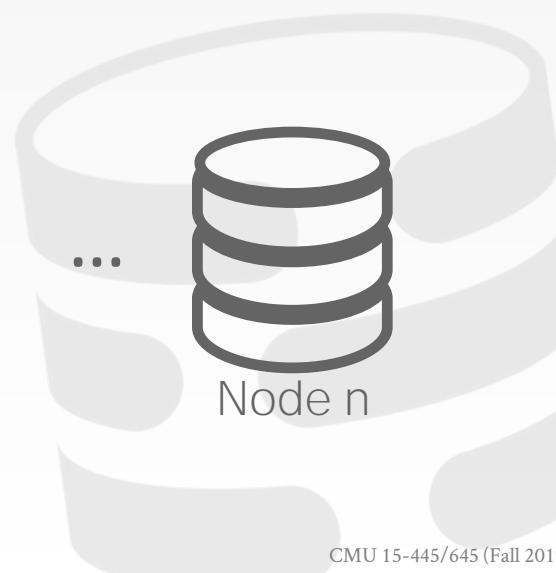
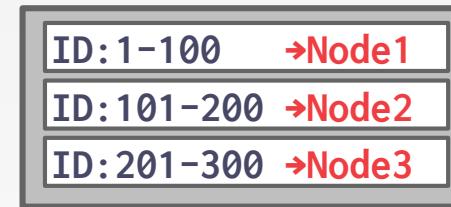
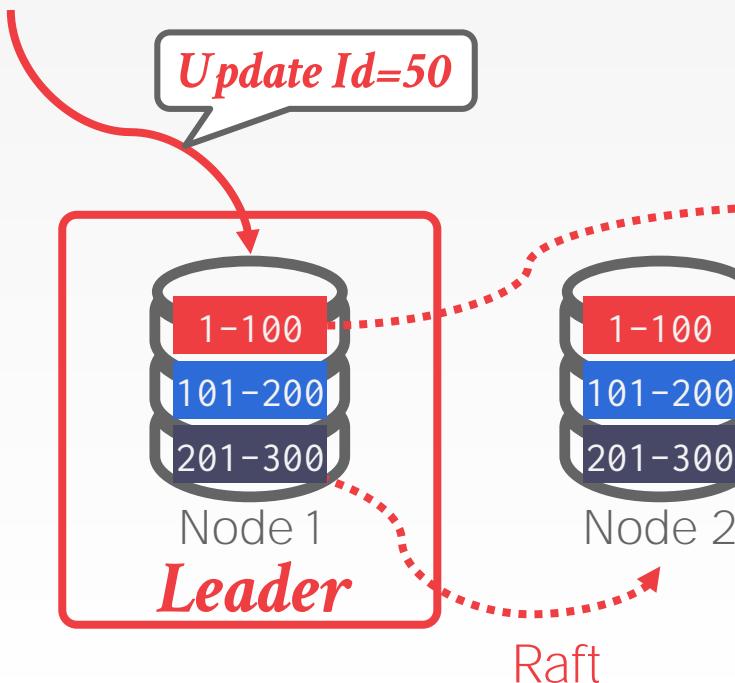


COCKROACHDB OVERVIEW



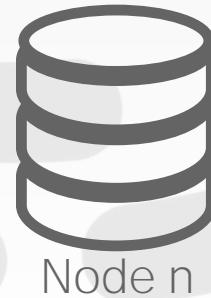
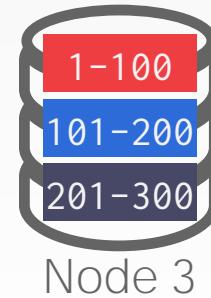
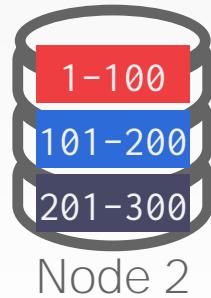
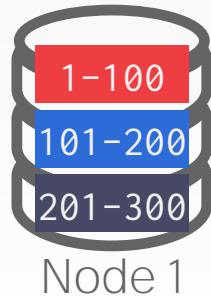
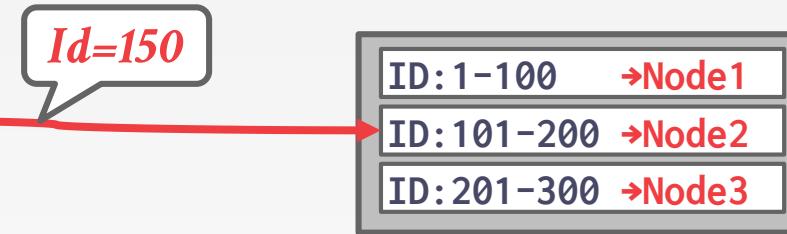


COCKROACHDB OVERVIEW



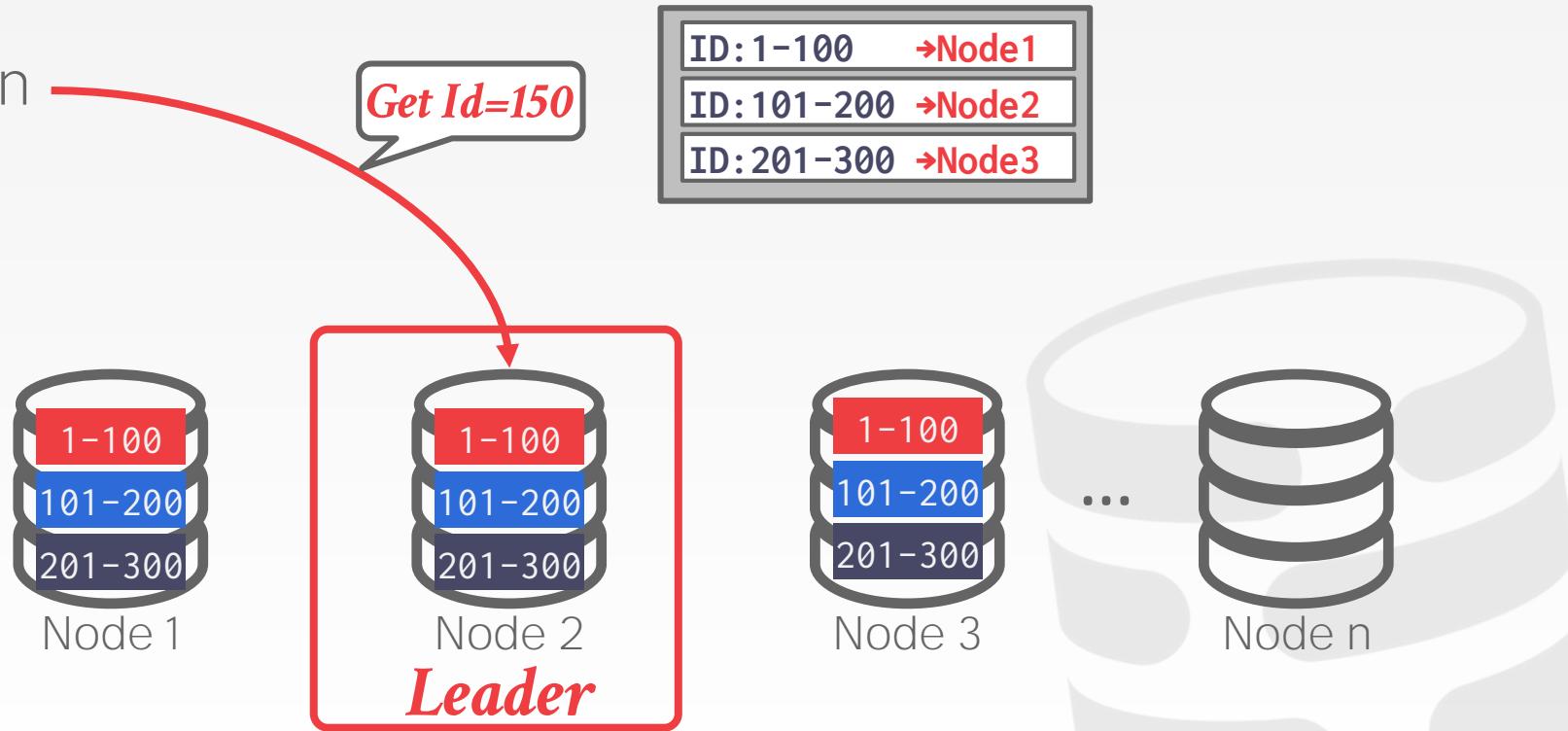


COCKROACHDB OVERVIEW





COCKROACHDB OVERVIEW



ANDY'S CONCLUDING REMARKS

Databases are awesome.

- They cover all facets of computer science.
- We have barely scratched the surface...

Going forth, you should now have a good understanding how these systems work.

This will allow you to make informed decisions throughout your entire career.

- Avoid premature optimizations.

