INTRODUCTION TO SQL

CS 564- Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

ANNOUNCEMENTS

- Enroll in Piazza!
- PS #1 will be posted tomorrow (due next Sunday)
- Group formation:
  - send an email to Elena with:
  - 3 **x** (Student IDs + emails)
  - only one person from every team!

WHAT IS THIS LECTURE ABOUT

- The Relational Model
- SQL: Basics
  - creating a table
  - primary keys
- SQL: Single-table queries
  - SELECT-FROM-WHERE structure
  - DISTINCT/ORDER BY/LIMIT
- SQL: Multi-table queries
  - foreign keys
  - joins

Relational Model

RELATIONAL MODEL

- first proposed by Codd in 1969
- has just a single concept: relation
- the world is represented as a collection of tables
- well-suited for efficient manipulations on computers

RELATION

The data is stored in **tables** (or relations)

table name

PRODUCT

attribute name

name	**category**	**price**	**manufacturer**
iPad	tablet	$399.00	Apple
Surface	tablet	$299.00	Microsoft
…	…	…	…

record/tuple

DOMAINS

- Each attribute has an atomic type called domain
- A **domain** specifies the set of values allowed
- Examples:
  - integer
  - string
  - real

PRODUCT(name: *string*,
category: *string*,
price: *real*,
manufacturer: *string*)

SCHEMA

The **schema** of a *relation*:

- relation name + attribute names
- **Product** (name, price, category, manufacturer)
- In practice we add the domain for each attribute

The **schema** of a *database:*

- a collection of relation schemas

INSTANCE

The **instance** of a *relation*:

– a set of tuples or records

The **instance** of a *database:*

– a collection of relation instances

EXAMPLE

PRODUCT(name: *string*,
category: *string*,
price: *real*,
manufacturer: *string*)

schema

instance

name	**category**	**price**	**manufacturer**
iPad	tablet	$399.00	Apple
Surface	tablet	$299.00	Microsoft
…	…	…	…

SCHEMA VS INSTANCE

- Analogy with programming languages:
  - schema ~ type
  - instance ~ value
- Important distinction
  - schema: stable over long periods of time
  - instance: changes constantly, as data is inserted/updated/deleted

SQL: BASICS

WHAT IS SQL?

- The most widely used database language
- Used to query and manipulate data

- SQL stands for **S**tructured **Q**uery **L**anguage
  - many SQL standards: SQL-92, SQL:1999, SQL:2011
  - vendors support different subsets
  - we will discuss the common functionality

CREATING A TABLE

table name

```
CREATE TABLE Author(
    authorid INTEGER PRIMARY KEY,
    firstname CHAR(20),
    lastname CHAR(30),
);
```

attributes

atomic types

PRIMARY KEYS

A **primary key** is a minimal subset of attributes that is a unique identifier of tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
- In SQL we specify that an attribute is the primary key with the keyword **`PRIMARY KEY`**

UNIQUE KEYS

- We can also define a unique key: a subset of attributes that uniquely defines a row:

```
CREATE TABLE Author(
    authorid INTEGER UNIQUE,
    firstname CHAR(20)) ;
```

- There can be only one primary key, but many unique keys!

NULL VALUES

- tuples in SQL relations can have `NULL` as a value for one or more attributes
- The meaning depends on context:
  - missing value: e.g. we know that Greece has some population, but we don't know what it is
  - inapplicable: e.g. the value of attribute *spouse* for an unmarried person

NULL VALUES

When creating a table in SQL, we can assert that a particular attribute takes no **NULL** values

```
CREATE TABLE Author(
        authorid INTEGER PRIMARY KEY,
        firstname CHAR(20) NOT NULL,
        lastname CHAR(30)
    );
```

POPULATING A TABLE

- To insert a single tuple:

  INSERT INTO *<relation>*

  VALUES *(<list of values>);*

- We may add to the relation name a list of attributes (if we forget the order)

```
INSERT INTO Author
VALUES(001, 'Dan’, 'Brown');
```

SQL: SINGLE-TABLE QUERIES

BASIC SQL QUERY

optional

SELECT [DISTINCT] attributes
FROM one or more tables
WHERE conditions on the tables

conditions of the form: Attr1 **op** Attr2

EXAMPLE

What is the population of USA?

```
SELECT Population
FROM Country
WHERE Code = 'USA';
```

PROJECTION: keeps only the specified attributes

SELECTION: filters the tuples of the relation

SEMANTICS

1. Think of a *tuple variable* ranging over each tuple of the relation mentioned in **`FROM`**
2. Check if the current tuple satisfies the **`WHERE`** clause
3. If so, compute the attributes or expressions of the **`SELECT`** clause using this tuple

* IN SELECT CLAUSES

When there is one relation in the **FROM** clause, * in the **SELECT** clause stands for *"all attributes of this relation"*

```
SELECT *
FROM City
WHERE Population >= '1000000'
AND CountryCode = 'USA';
```

RENAMING ATTRIBUTES

If we want the output schema to have different attribute names, we can use **AS** *<new name>* to rename an attribute

```
SELECT Name AS LargeUSACity
FROM City
WHERE Population >= '1000000'
AND CountryCode = 'USA';
```

ARITHMETIC EXPRESSIONS

We can use any arithmetic expression (that makes sense) in the **SELECT** clause

```
SELECT Name,
 (Population/ 1000000) AS PopulationInMillion
FROM City
WHERE Population >= '1000000’ ;
```

WHAT CAN WE USE IN WHERE CLAUSES?

- attribute names of the relations that appear in the `FROM` clause
- comparison operators: `=, <>, <, >, <=, >=`
- arithmetic operations (+, -, /, *)
- **AND**, **OR**, **NOT** to combine conditions
- operations on strings (e.g. concatenation)
- pattern matching: *s* `LIKE` *p*
- special functions for comparing dates and times

PATTERN MATCHING

s **LIKE** p: pattern matching on strings

- % = any sequence of characters
- _ = any single character

```
SELECT Name, GovernmentForm
FROM Country
WHERE GovernmentForm LIKE '%Monarchy%';
```

USING DISTINCT

- The default semantics of SQL is **bag** semantics (duplicate tuples are allowed in the output)
- The use of **DISTINCT** in the **SELECT** clause removes all duplicate tuples in the result, and returns a **set**

```
SELECT DISTINCT GovernmentForm
FROM Country;
```

ORDER BY

The use of **ORDER BY** orders the tuples by the attribute we specify in decreasing (**DESC**) or increasing (**ASC**) order

```
SELECT Name, Population
FROM City
WHERE Population >= '1000000'
ORDER BY Population DESC;
```

LIMIT

- The use of **LIMIT <*number*>** limits the output to be only the specified number of tuples
- It can be used with **ORDER BY** to get the maximum or minimum value of an attribute!

```
SELECT Name, Population
FROM City
ORDER BY Population DESC
LIMIT 2;
```

SQL: MULTI-TABLE QUERIES

FOREIGN KEYS

Suppose that we want to create a table Book, and make sure that *the author of the book exists in the table* Author

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid));
```

FOREIGN KEYS

- Use the keyword **REFERENCES**, as:

FOREIGN KEY *(<list of attributes>)*

REFERENCES *<relation> (<attributes>)*

- Referenced attributes must be declared **PRIMARY KEY** or **UNIQUE**

ENFORCING FK CONSTRAINTS

If there is a foreign-key constraint from attributes of relation $\boldsymbol{R}$ to the primary key of relation $\boldsymbol{S}$, two violations are possible:

1. An insert or update to $\boldsymbol{R}$ introduces values not found in $\boldsymbol{S}$
2. A deletion or update to $\boldsymbol{S}$ causes some tuples of $\boldsymbol{R}$ to dangle

There are 3 ways to enforce foreign key constraints!

ACTION 1: REJECT

- The insertion/deletion/update query is rejected and not executed in the DBMS
- This is the default action if a foreign key constraint is declared

ACTION 2: CASCADE UPDATE

When a tuple referenced is *updated*, the update propagates to the tuples that reference it

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid)
   ON UPDATE CASCADE);
```

ACTION 2: CASCADE DELETE

When a tuple referenced is *deleted*, the deletion propagates to the tuples that reference it

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid)
   ON DELETE CASCADE);
```

ACTION 3: SET NULL

- When a delete/update occurs, the values that reference the deleted tuple are set to **NULL**

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid)
   ON UPDATE SET NULL);
```

WHAT SHOULD WE CHOOSE?

- When we declare a foreign key, we may choose policies `SET NULL` or **CASCADE** *independently* for deletions and updates

  ON [UPDATE, DELETE] [SET NULL, CASCADE]

- Otherwise, the default policy (*reject*) is used

MULTIPLE RELATIONS

- We often want to combine data from more than one relation
- We can address several relations in one query by listing them all in the `FROM` clause
- If two attributes from different relations have the same name, we can distinguish them by writing *<relation>***.***<attribute>*

EXAMPLE

What is the name of countries that speak Greek?

```
SELECT Name
FROM Country, CountryLanguage
WHERE Code = CountryCode
   AND Language = 'Greek';
```

This is **BAD** style!!

EXAMPLE: GOOD STYLE

```
SELECT Country.Name
FROM Country, CountryLanguage
WHERE Country.Code=CountryLanguage.CountryCode
AND CountryLanguage.Language = 'Greek';
```

```
SELECT C.Name
FROM Country C, CountryLanguage L
WHERE C.Code = L.CountryCode
AND L.Language = 'Greek';
```

VARIABLES

Variables are necessary when we want to use two copies of the same relation in the **FROM** clause

```
SELECT C.Name
FROM Country C, CountryLanguage L1,
CountryLanguage L2
WHERE  C.Code = L1.CountryCode
   AND C.Code = L2.CountryCode
   AND L1.Language = 'Greek’
   AND L2.Language = 'English';
```

SEMANTICS: SELECT-FROM-WHERE

1. Start with the cross product of all the relations in the **`FROM`** clause
2. Apply the conditions from the **`WHERE`** clause
3. Project onto the list of attributes and expressions in the **`SELECT`** clause
4. If **`DISTINCT`** is specified, eliminate duplicate rows

SEMANTICS OF SQL: EXAMPLE

```
SELECT R.D
FROM   R, S
WHERE  R.A = S.B AND S.C = 'e' ;
```

A	D
1	a
2	b
2	c

B	C
1	d
2	e

cross product →

A	D	B	C
1	a	1	d
1	a	2	e
2	b	1	d
2	b	2	e
2	c	1	d
2	c	2	e

select →

A	D	B	C
2	b	2	e
2	c	2	e

project ↓

D
b
c

SEMANTICS OF SQL: NESTED LOOP

```
SELECT a1, a2, …, ak
FROM   R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

answer := {}
for x_1 **in** R_1 **do**
    for x_2 **in** R_2 **do**
        …..
            for x_n **in** R_n **do**
                if Conditions
                    then *answer* := *answer* ∪ $\{(a_1,\ldots,a_k)\}$
return *answer*

SEMANTICS OF SQL

- The query processor will almost never evaluate the query this way
- SQL is a declarative language
- The DBMS figures out the most efficient way to compute it (we will discuss this later in the course when we talk about *query optimization*)

ADVANCED SQL I

CS 564- Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

CLASS ANNOUNCEMENTS

- Last day to declare your group is today!
- **Sunday**: deadline for PS 1
- Peer mentor office hours change for this week:
  - Wednesday 4:00 - 6:30 changes to
    Wednesday 4:00 - 5:30 + Friday 5:00 - 6:30

WHAT IS THIS LECTURE ABOUT

- SQL: Set Operators
  - UNION/EXCEPT/INTERSECT
  - duplicates in SQL
- SQL: Nested Queries
  - IN/EXISTS/ALL
  - correlated queries

Set and Multiset Operators

SET OPERATORS: REFRESHER

$$R = \{1, 2, 3\} \qquad S = \{1, 2, 4, 5\}$$

- Intersection: $R \cap S = \{1, 2\}$
- Union: $R \cup S = \{1, 2, 3, 4, 5\}$
- Difference: $R - S = \{3\}$

  $S - R = \{4, 5\}$

SET OPERATORS IN SQL

SQL supports set operations between the outputs of subqueries:

- (subquery) **`INTERSECT`** (subquery)
- (subquery) **`UNION`** (subquery)
- (subquery) **`EXCEPT`** (subquery)

SET OPERATORS: INTERSECT

```
SELECT A FROM R
INTERSECT
SELECT A FROM S;
```

R

A
1
1
1
2
3

S

A
1
1
2
2
4
5

output

A
1
2

Returns the tuples that belong in both subquery results

SET OPERATORS: UNION

```
SELECT A FROM R
UNION
SELECT A FROM S;
```

R

A
1
1
1
2
3

S

A
1
1
2
2
4
5

output

A
1
2
3
4
5

Returns the tuples that belong in either subquery results

SET OPERATORS: EXCEPT

```
SELECT A FROM R
EXCEPT
SELECT A FROM S;
```

R

A
1
1
1
2
3

S

A
1
1
2
2
4
5

output

A
3

Returns the tuples that belong in the first and not the second subquery result

SEMANTICS

- When using set operators, SQL eliminates all duplicate tuples
- We can modify the semantics by using the keyword **`ALL`** (e.g. **`UNION ALL`**)
- When using **`ALL`**, the operators are evaluated using multiset (or bag) semantics

SET OPERATORS: UNION ALL

```
SELECT A FROM R
UNION ALL
SELECT A FROM S;
```

output

A
1
1
1
1
1
2
2
2
3
4
5

R

A
1
1
1
2
3

S

A
1
1
2
2
4
5

The number of copies of each tuple is the sum of the number of copies in the subqueries

SET OPERATORS: INTERSECT ALL

```
SELECT A FROM R
INTERSECT ALL
SELECT A FROM S;
```

R

A
1
1
1
2
3

S

A
1
1
2
2
4
5

output

A
1
1
2

The number of copies of each tuple is the minimum of the number of copies in the subqueries

SET OPERATORS: EXCEPT ALL

```
SELECT A FROM R
EXCEPT ALL
SELECT A FROM S;
```

R

A
1
1
1
2
3

S

A
1
1
2
2
4
5

output

A
1
3

The number of copies of each tuple is the difference (if positive) of the number of copies in the subqueries

DISCUSSION ON DUPLICATES

- When doing projection:
  - easier to avoid eliminating duplicates
  - *tuple-at-a-time* processing
- When doing intersection, union or difference:
  - algorithms typically sort the relations first
  - at that point you may as well eliminate the duplicates anyway

NESTED QUERIES

NESTED QUERIES

A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places:

- in **FROM** clauses
- in **WHERE** clauses

outer query

```
SELECT C.Name
FROM Country C
WHERE C.code =
    (SELECT C.CountryCode
     FROM City C
     WHERE C.name = 'Berlin');
```

inner query

NESTING

- We can write nested queries because the SQL language is compositional
- Everything is represented as a multiset
- Hence the output of one query can be used as the input to another (nesting)

NESTED QUERIES

Find all countries in Europe with population more than 50 million

```
SELECT C.Name
FROM (SELECT Name, Continent
       FROM Country
       WHERE Population >50000000) AS C
WHERE C.Continent = 'Europe' ;
```

UNNESTING

Unnesting means to find an equivalent SQL query that does not use nesting!

```
SELECT C.Name
FROM (SELECT Name, Continent
       FROM Country
       WHERE Population >50000000) AS C
WHERE C.Continent = 'Europe' ;
```

unnesting

```
SELECT Name
FROM Country
WHERE Population > 50000000
  AND Continent = 'Europe' ;
```

SET-COMPARISON OPERATOR: IN

Find all countries in Europe that have ***some*** *city with population more than 5 million*

```
SELECT C.Name
FROM Country C
WHERE C.Continent = 'Europe'
AND C.Code IN (SELECT CountryCode
               FROM City
               WHERE Population > 5000000);
```

SET-COMPARISON OPERATOR: EXISTS

Find all countries in Europe that have ***some*** *city with population more than 5 million*

```
SELECT C.Name
FROM Country C
WHERE C.Continent = 'Europe’
AND EXISTS (SELECT *
            FROM City T
            WHERE T.Population > 5000000
            AND T.CountryCode = C.Code);
```

CORRELATED SUBQUERIES

- A correlated subquery uses values defined in the outer query
- The inner subquery gets executed multiple times!

```
SELECT C.Name
FROM Country C
WHERE C.Continent = 'Europe'
AND EXISTS (SELECT *
            FROM City T
            WHERE T.Population > 5000000
            AND T.CountryCode = C.Code);
```

correlated subquery

SET-COMPARISON OPERATOR: ANY

*Find all countries in Europe that have **some** city with population more than 5 million*

```
SELECT C.Name
FROM Country C
WHERE C.Continent = 'Europe’
AND 5000000 <= ANY (SELECT T.Population
          FROM City T
          WHERE T.CountryCode = C.Code);
```

The operator before **ANY** must be a comparison operator!

SET-COMPARISON OPERATORS

*Find all countries in Europe that have **all** cities with population less than 1 million*

```
SELECT C.Name
FROM Country C
WHERE C.Continent = 'Europe’
AND NOT EXISTS (SELECT *
                FROM City T
                WHERE T.Population > 1000000
                AND T.CountryCode = C.Code);
```

SET-COMPARISON OPERATORS: ALL

*Find all countries in Europe that have **all** cities with population less than 1 million*

```
SELECT C.Name
FROM Country C
WHERE C.Continent = 'Europe’
AND 1000000 > ALL (SELECT T.Population
          FROM City T
          WHERE T.CountryCode = C.Code);
```

ADVANCED SQL II

CS 564 - Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT

- SQL: Aggregation
  - Aggregate operators
  - GROUP BY
  - HAVING
- SQL: Nulls
- SQL: Outer Joins

AGGREGATION

AGGREGATION

- **SUM,AVG,COUNT,MIN,MAX** can be applied to a column in a **SELECT** clause to produce that aggregation on the column
- **COUNT(*)** simply counts the number of tuples

```
SELECT AVG(Population)
FROM Country
WHERE Continent = 'Europe';
```

AGGREGATION: ELIMINATE DUPLICATES

We can use **COUNT**(DISTINCT <attribute>) to remove duplicate tuples before counting!

```
SELECT COUNT (DISTINCT Language)
FROM CountryLanguage ;
```

GROUP BY

- We may follow a **SELECT-FROM-WHERE** expression by **GROUP BY** and a list of attributes
- The relation is then grouped according to the values of those attributes, and any aggregation is applied only within each group

```
SELECT Continent, COUNT(*)
FROM Country
GROUP BY Continent;
```

GROUP BY: EXAMPLE

```
SELECT A, SUM(B * C)
FROM R
GROUP BY A;
```

5 = 2*0 + 5*1

R

A	B	C
a	2	0
a	5	1
b	7	1
b	6	0
c	4	1

grouping

A	B	C
a	2	0
	5	1
b	7	1
	6	0
c	4	1

select clause

A	SUM(B*C)
a	5
b	7
c	4

RESTRICTIONS

If any aggregation is used, then each element of the **SELECT** list must be either:

- aggregated, or
- an attribute on the **GROUP BY** list

This query is **wrong**!!

```
SELECT Continent, COUNT(Code)
FROM Country
GROUP BY Code;
```

GROUP BY + HAVING

- The **HAVING** clause always follows a **GROUP BY** clause in a SQL query
  - it applies to each group, and groups not satisfying the condition are removed
  - it can refer only to attributes of relations in the **FROM** clause, as long as the attribute makes sense within a group

The HAVING clause applies **only** on aggregates!

HAVING: EXAMPLE

```
SELECT Language, COUNT(CountryCode) AS N
FROM CountryLanguage
WHERE Percentage >= 50
GROUP BY Language
HAVING N > 2
ORDER BY N DESC ;
```

PUTTING IT ALL TOGETHER

```
SELECT [DISTINCT] S
FROM R, S, T ,…
WHERE C1
GROUP BY attributes
HAVING C2
ORDER BY attribute ASC/DESC
LIMIT N ;
```

CONCEPTUAL EVALUATION

1. Compute the **`FROM-WHERE`** part, obtain a table with all attributes in R,S,T,…
2. Group the attributes in the **`GROUP BY`**
3. Compute the aggregates and keep only groups satisfying condition C2 in the **`HAVING`** clause
4. Compute aggregates in S
5. Order by the attributes specified in **`ORDER BY`**
6. Limit the output if necessary

NULL VALUES

NULL VALUES

- tuples in SQL relations can have `NULL` as a value for one or more attributes
- The meaning depends on context:
  - Missing value: *e.g.* we know that Greece has some population, but we don't know what it is
  - Inapplicable: *e.g.* the value of attribute *spouse* for an unmarried person

NULL PROPAGATION

- When we do arithmetic operations using **NULL**, the result is again a **NULL**
  - (10 * x)+5 returns **NULL** if x = **NULL**
  - **NULL**/0 also returns **NULL**!

- String concatenation also results in **NULL** when one of the operands is **NULL**
  - 'Wisconsin' || **NULL**|| '-Madison’ returns **NULL**

COMPARISONS WITH NULL

- The logic of conditions in SQL is 3-valued logic:
  - **TRUE** = 1
  - **FALSE** = 0
  - **UNKNOWN** = 0.5
- When any value is compared with a **`NULL`**, the result is **UNKNOWN**
  - *e.g.* x > 5 is **UNKNOWN** if x = **`NULL`**
- A query produces a tuple in the answer only if its truth value in the **`WHERE`** clause is **TRUE** (1)

3-VALUED LOGIC

The truth value of a **WHERE** clause is computed using the following rules:

- C1 **AND** C2 ----> *min*{ value(C1), value(C2) }
- C1 **OR** C2 ----> *max*{ value(C1), value(C2) }
- **NOT** C ----> 1- value(C)

3-VALUED LOGIC: EXAMPLE

tuple (1, NULL, NULL)

```
SELECT *
FROM R
WHERE (R.A>0) AND ((R.B<5) OR (NOT R.C=3));
```

1

0.5

0.5

0.5 (1-0.5)

0.5 (max{0.5, 0.5})

0.5 (min{0.5, 1})

the expression is **UNKNOWN**!

COMPLICATIONS

What will happen in the following query?

```
SELECT COUNT(*)
FROM Country
WHERE IndepYear > 1990 OR IndepYear <= 1990 ;
```

It will not count the rows with NULL!

TESTING FOR NULL

We can test for **NULL** explicitly:

- x **IS** NULL
- x **IS** NOT NULL

```
SELECT COUNT(*)
FROM Country
WHERE IndepYear > 1990 OR IndepYear <= 1990
OR IndepYear IS NULL;
```

OUTER JOINS

INNER JOINS

The joins we have seen so far are inner joins

```
SELECT C.Name AS Country, MAX(T.Population) AS N
FROM Country C, City T
WHERE C.Code = T.CountryCode
GROUP BY C.Name;
```

Alternative syntax:

```
SELECT C.Name AS Country, MAX(T.Population) AS N
FROM Country C
INNER JOIN  City T ON C.Code = T.CountryCode
GROUP BY C.Name;
```

We can simply also write **JOIN**

LEFT OUTER JOINS

A left outer join includes tuples from the left relation even if there's no match on the right! It fills the remaining attributes with NULL

```
SELECT C.Name AS Country, MAX(T.Population)
FROM Country C
LEFT OUTER JOIN City T
   ON C.Code = T.CountryCode
GROUP BY C.Name ;
```

LEFT OUTER JOIN: EXAMPLE

```
SELECT A, C
FROM R LEFT OUTER JOIN S
ON R.B = S.B
```

R

A	B
a	2
a	5
b	5
c	6

S

B	C
2	100
3	200
5	300
7	400

→

A	C
a	100
a	300
b	300
c	**NULL**

OTHER OUTER JOINS

- Left outer join:
  - include the left tuple even if there is no match
- Right outer join:
  - include the right tuple even if there is no match
- Full outer join:
  - include the both left and right tuples even if there is no match

ENTITY-RELATIONSHIP MODEL

CS 564- Spring 2020

WHAT IS THIS LECTURE ABOUT

E/R Model:

- entity sets, attribute
- relation: binary, multi-way
- relationship roles, attributes on relationships
- subclasses (ISA)
- weak entity sets
- constraints
- design principles
- E/R to Relational Model

HOW TO BUILD A DB APPLICATION

HOW TO BUILD A DB APPLICATION

- Pick an application
- Figure out what to model (ER model)
  - Output: ER diagram
- Transform the ER diagram to a relational schema
- Refine the relational schema (normalization)
- Now ready to implement the schema and load the data!

RUNNING EXAMPLE

We want to store information about:

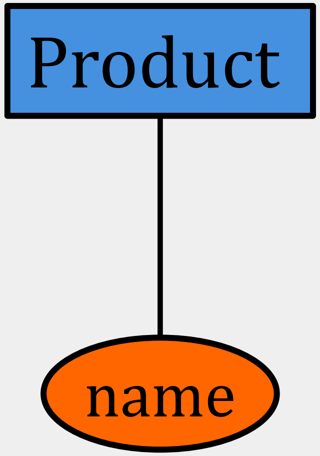
- companies and employees
  - Each **company** has a name, an address, …
  - Each company has a list of **employees**
- products manufactured by these companies
  - Each **product** has a name, a description, …

E/R MODEL

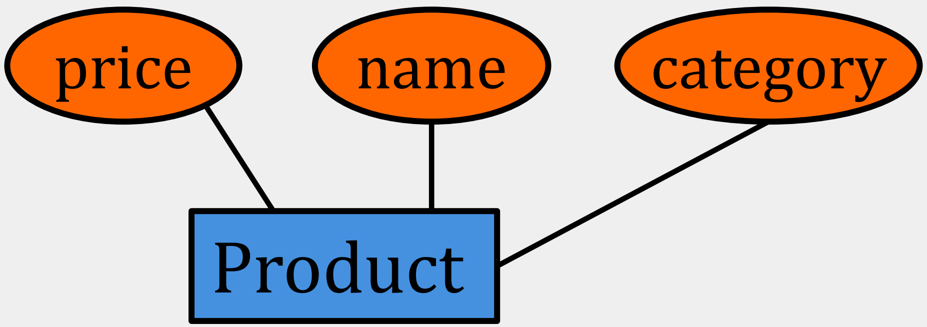
- Gives us a **visual language** to specify
  - what information the DB must hold
  - what are the relationships among components of that information
- Proposed by Peter Chen in 1976
- What we will cover:
  1. basic stuff: entities, attributes, relationships
  2. constraints
  3. weak entity sets
  4. design principles

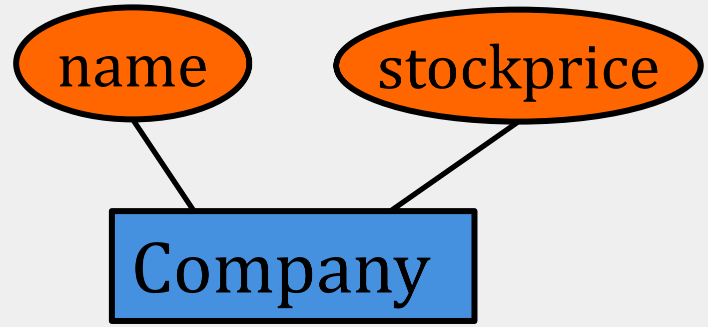
ENTITIES & ATTRIBUTES

- **Entity**
  - an object distinguishable from other object
- **Entity set**
  - a collection of similar entities
  - represented by rectangles
  - described using a set of attributes
- **Attribute**
  - represented by ovals attached to an entity set

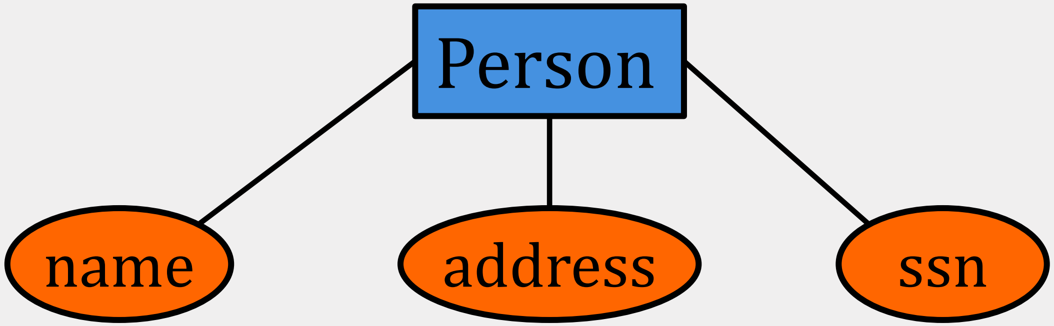


ENTITY SETS & ATTRIBUTES



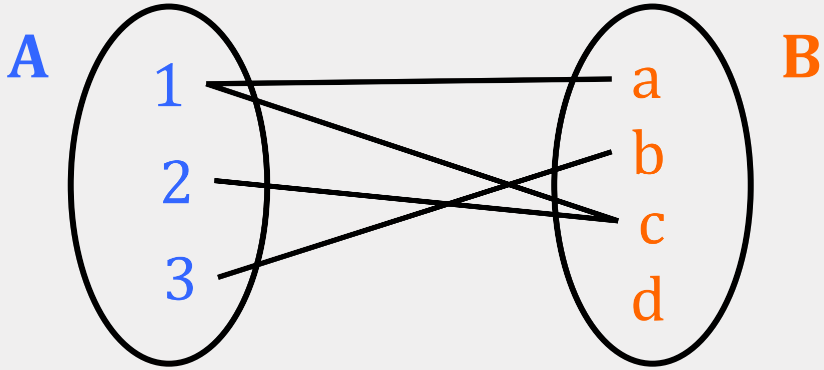


Entities are not explicitly represented in E/R diagrams!

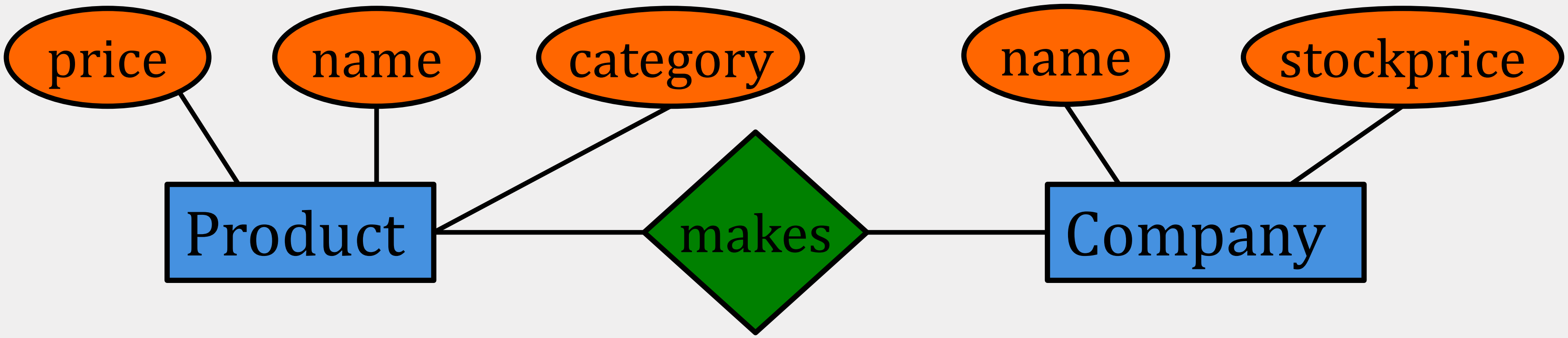


RELATIONS

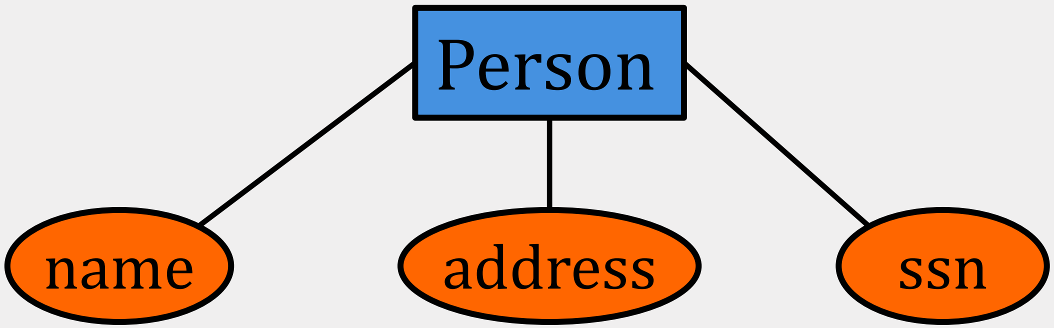
- A mathematical definition:
  - if **A**, **B** are sets, then a **<u>relation</u>** **R** is a subset of **A** x **B**
- Example
  - **A** = {1, 2, 3}, **B** = {a, b, c, d}
  - **R** = {(1, a), (1, c), (2, c), (3, b)}



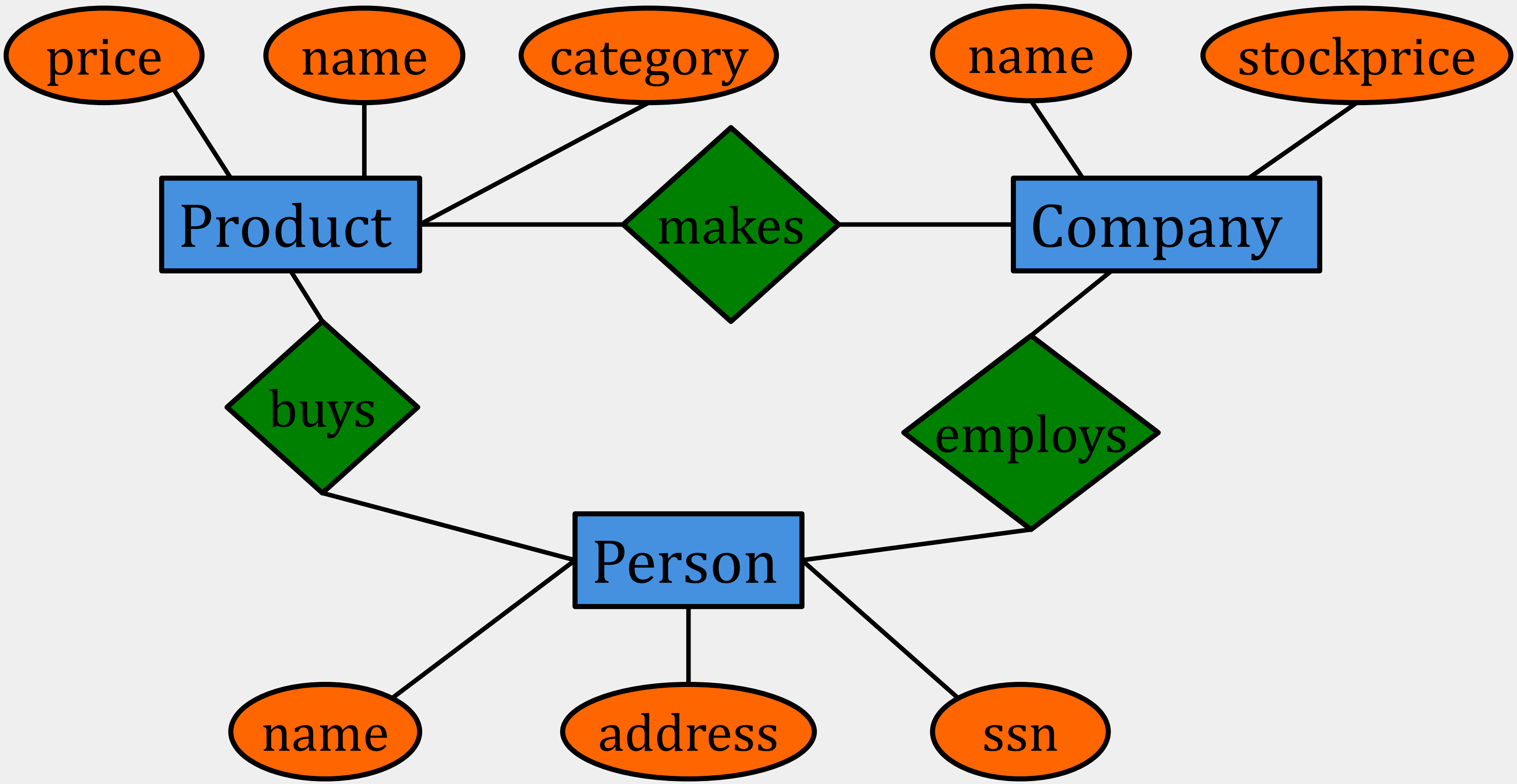
RELATIONSHIPS



makes is a subset of **Product** x **Company**

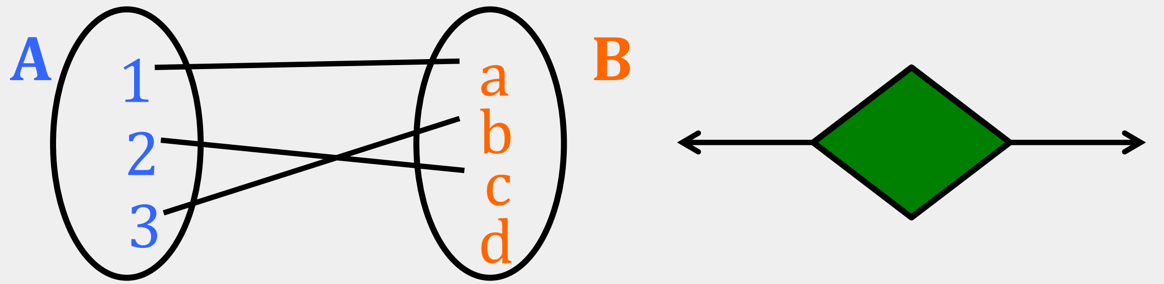


RELATIONSHIPS

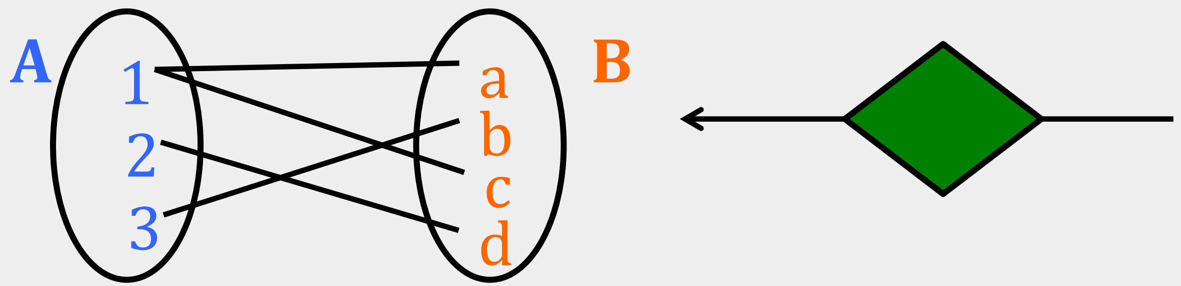


MULTIPLICITY OF RELATIONSHIPS

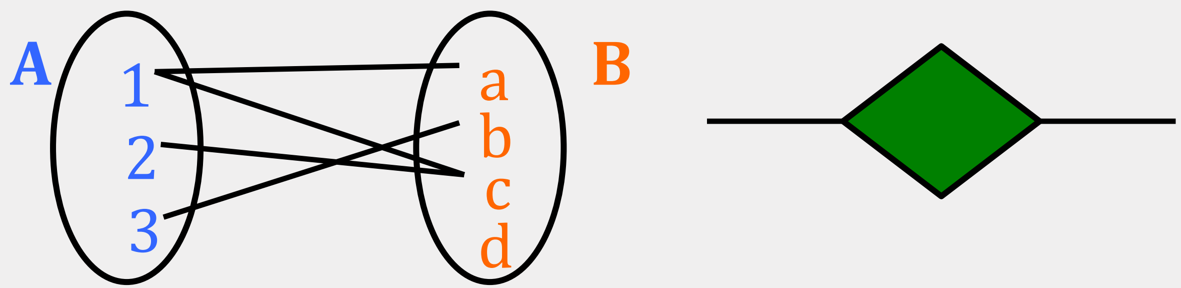
- one-one



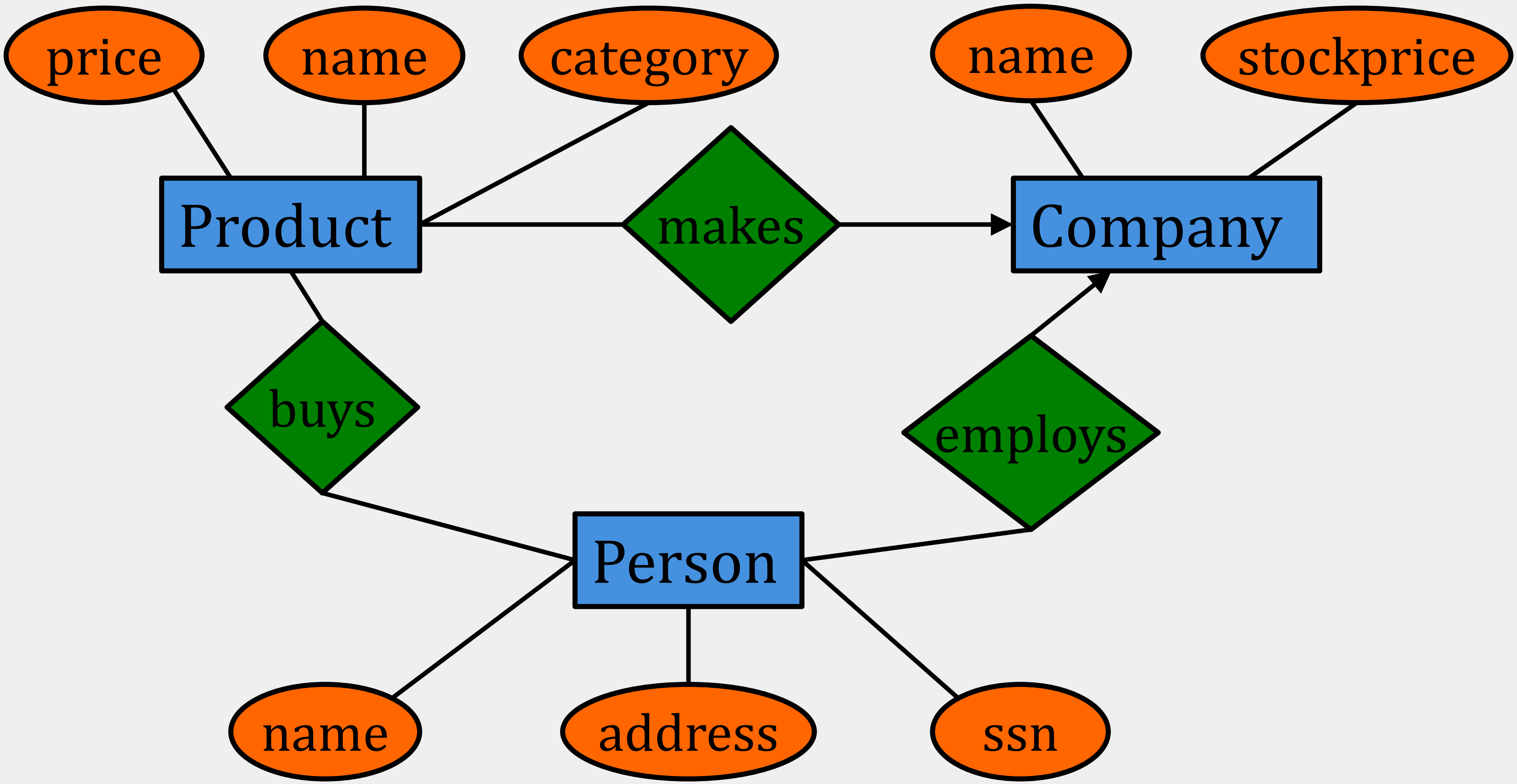
- many-one



- many-many

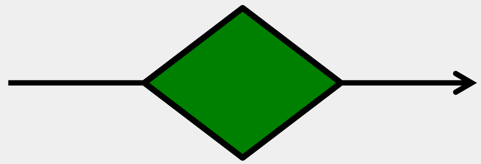


MULTIPLICITY OF RELATIONSHIPS

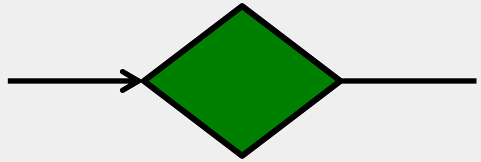


NOTATION DIFFERENCE

- We use:



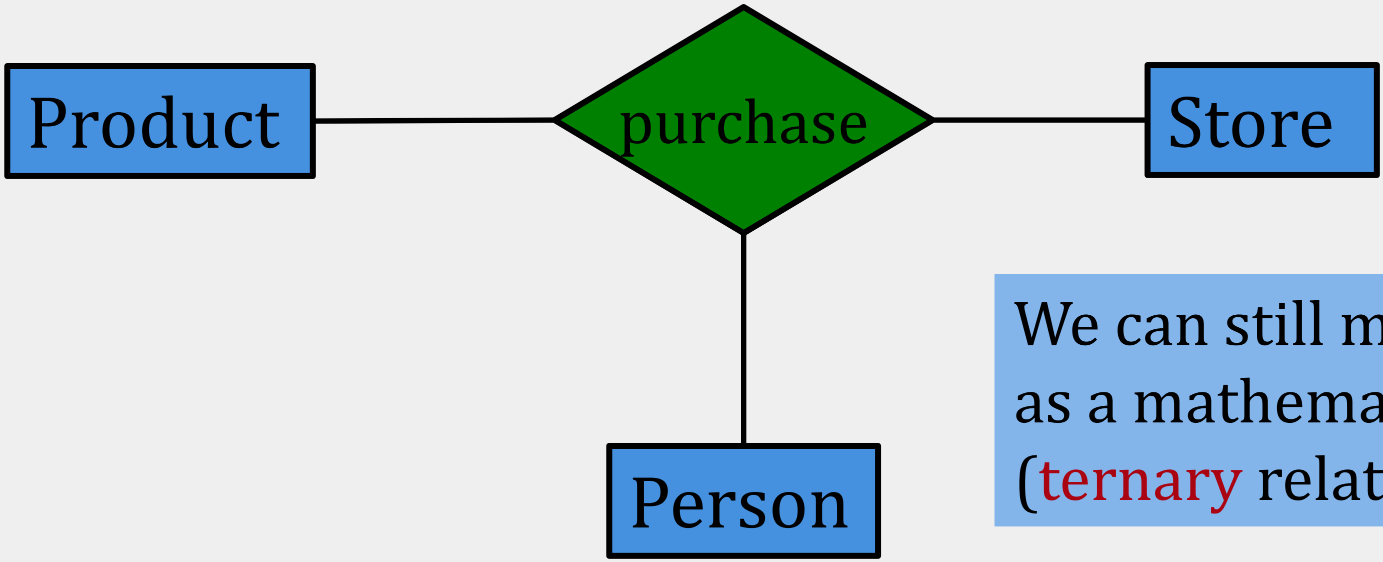
- The cow book uses (page 33):



You should use the notation in the slides!

MULTI-WAY RELATIONSHIPS

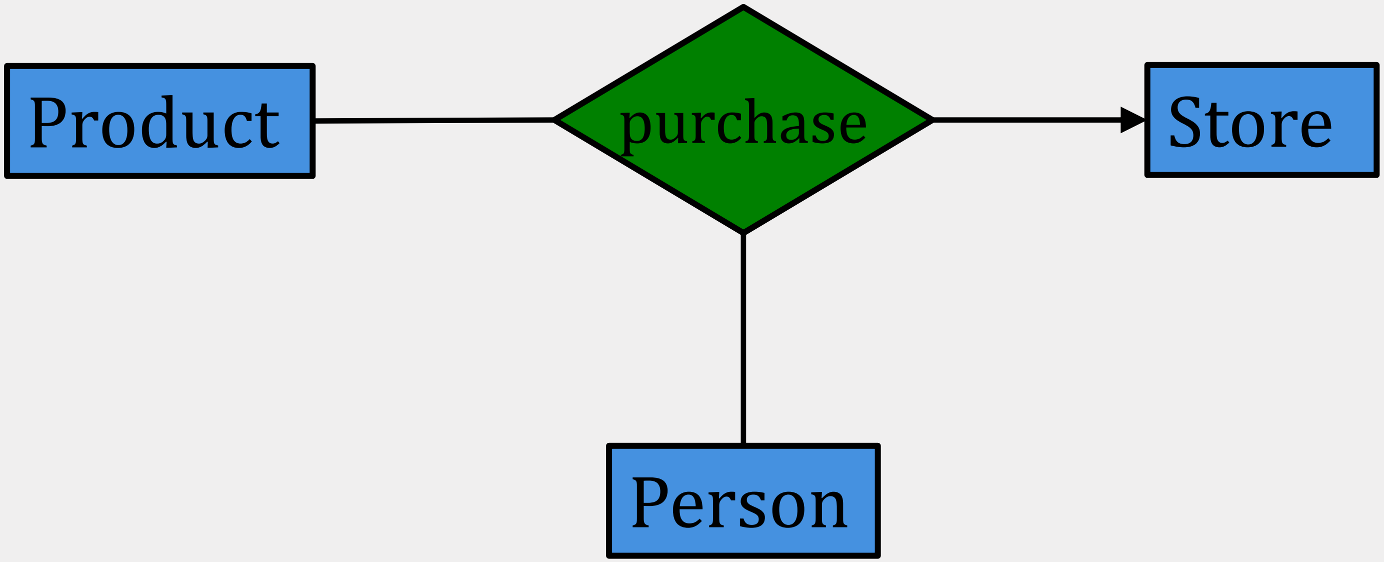
How do we model a **purchase** relation between **buyers**, **products** and **stores**?



We can still model this as a mathematical set (ternary relation)

ARROWS IN MULTI-WAY RELATIONSHIPS

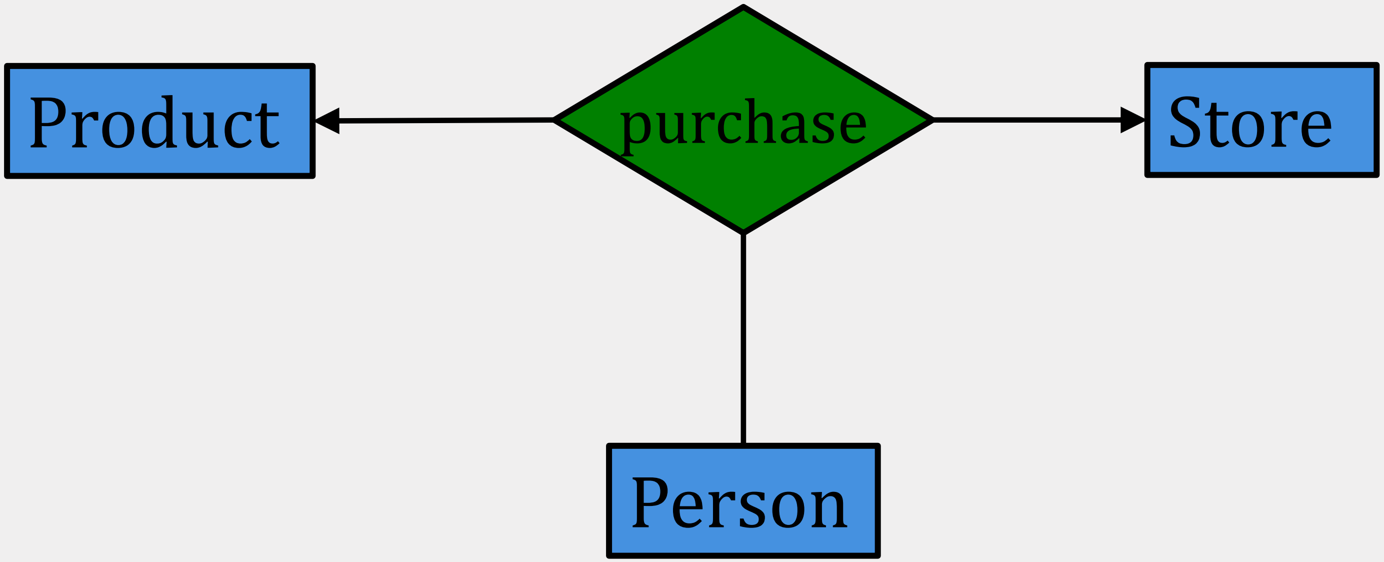
What does the arrow mean here?



A given **person** can **purchase** a given **product** from at most one **store**!

ARROWS IN MULTI-WAY RELATIONSHIPS

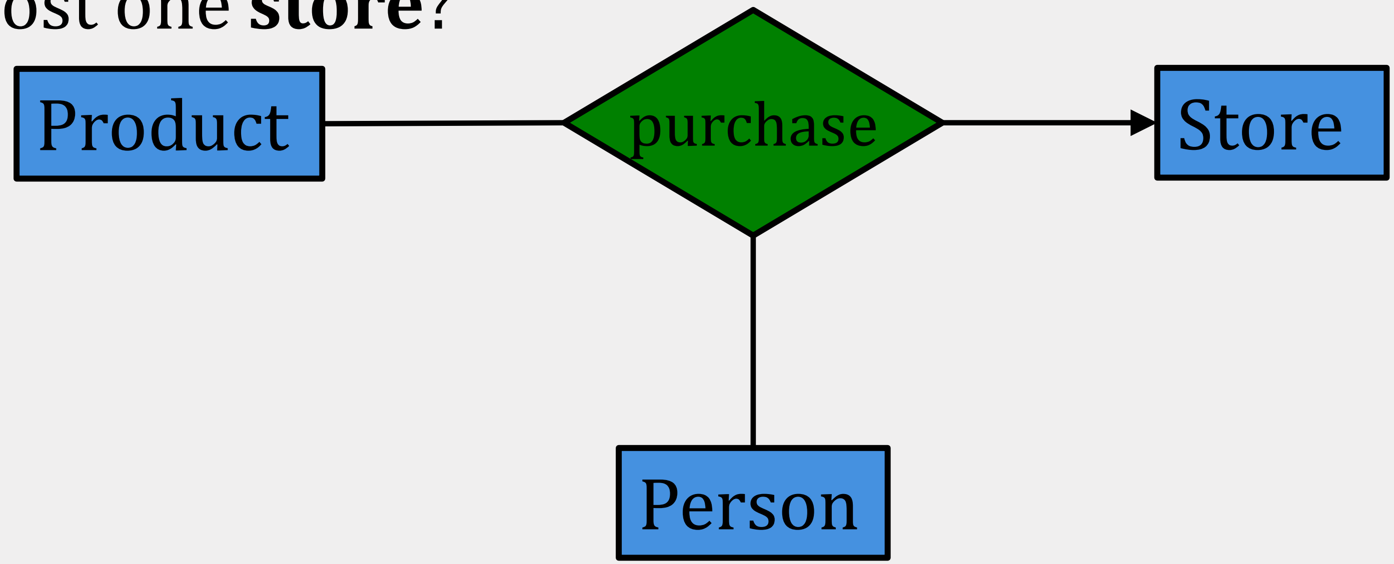
What about here?



A given **person** can **purchase** a given **product** from at most one **store** *AND* a given **store** sells to a given **person** at most one **product**

ARROWS IN MULTI-WAY RELATIONSHIPS

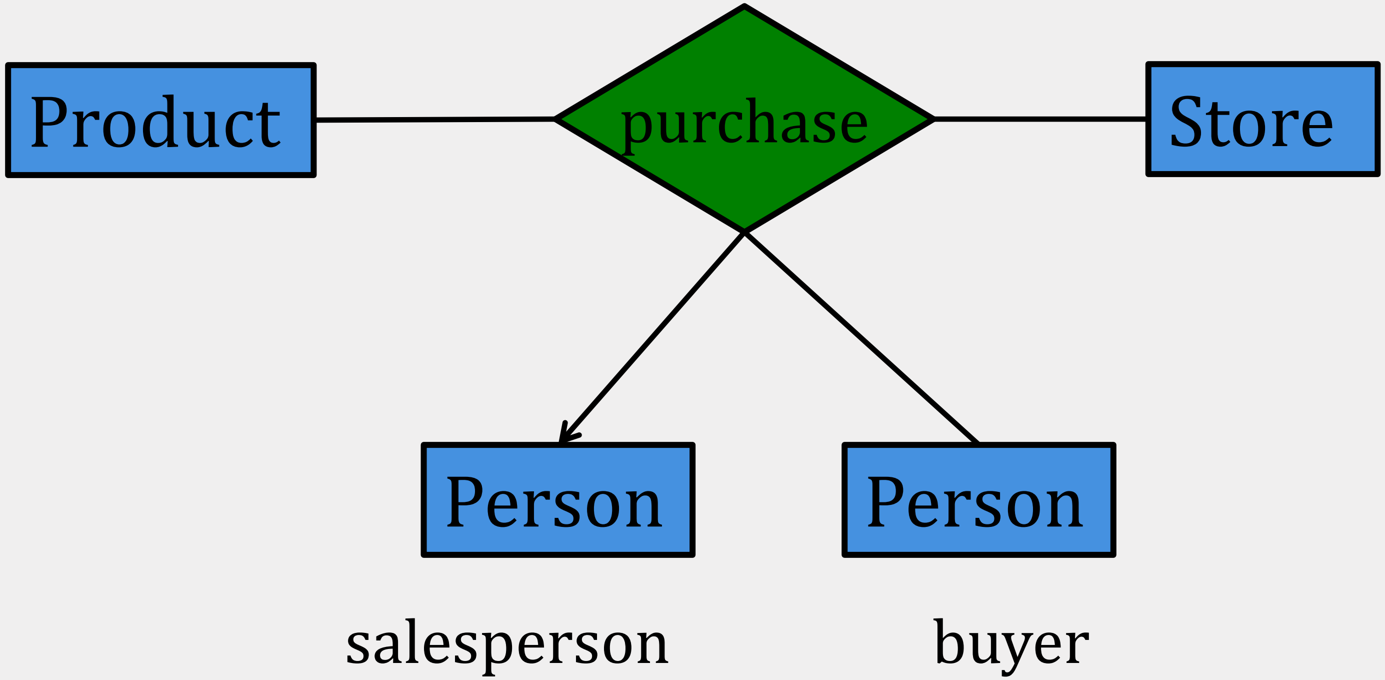
How can we say that a given **person** buys from at most one **store**?



Not possible, we can only approximate!

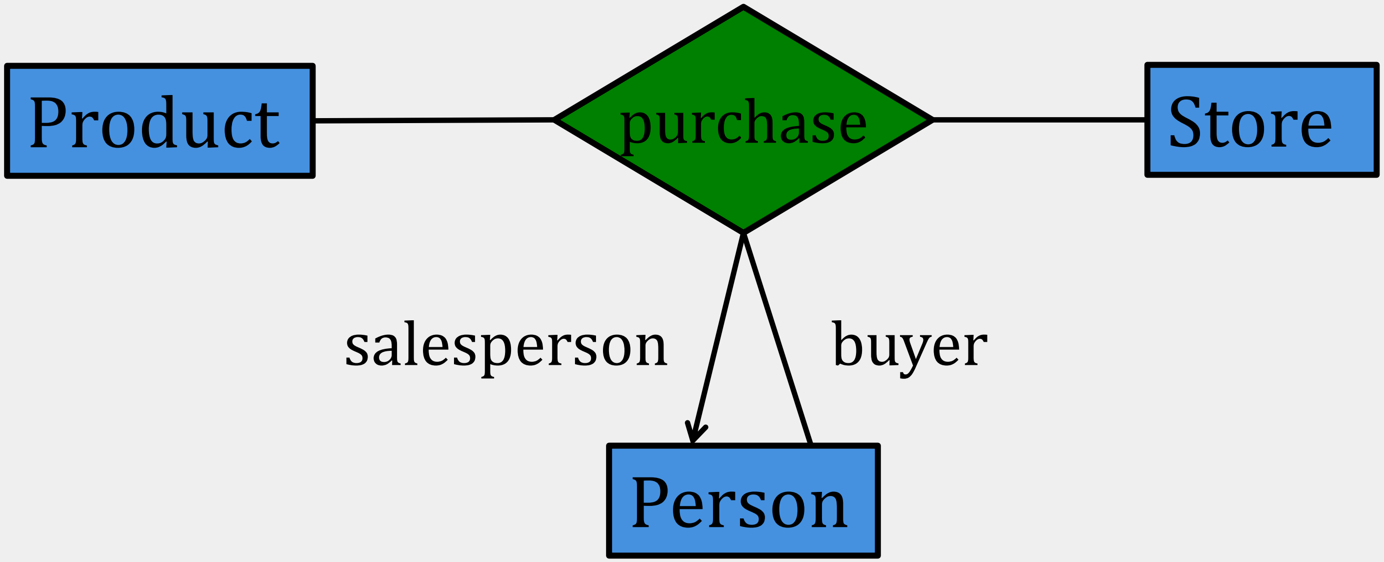
ROLES IN RELATIONSHIPS

What if we need an entity set twice in a relationship?

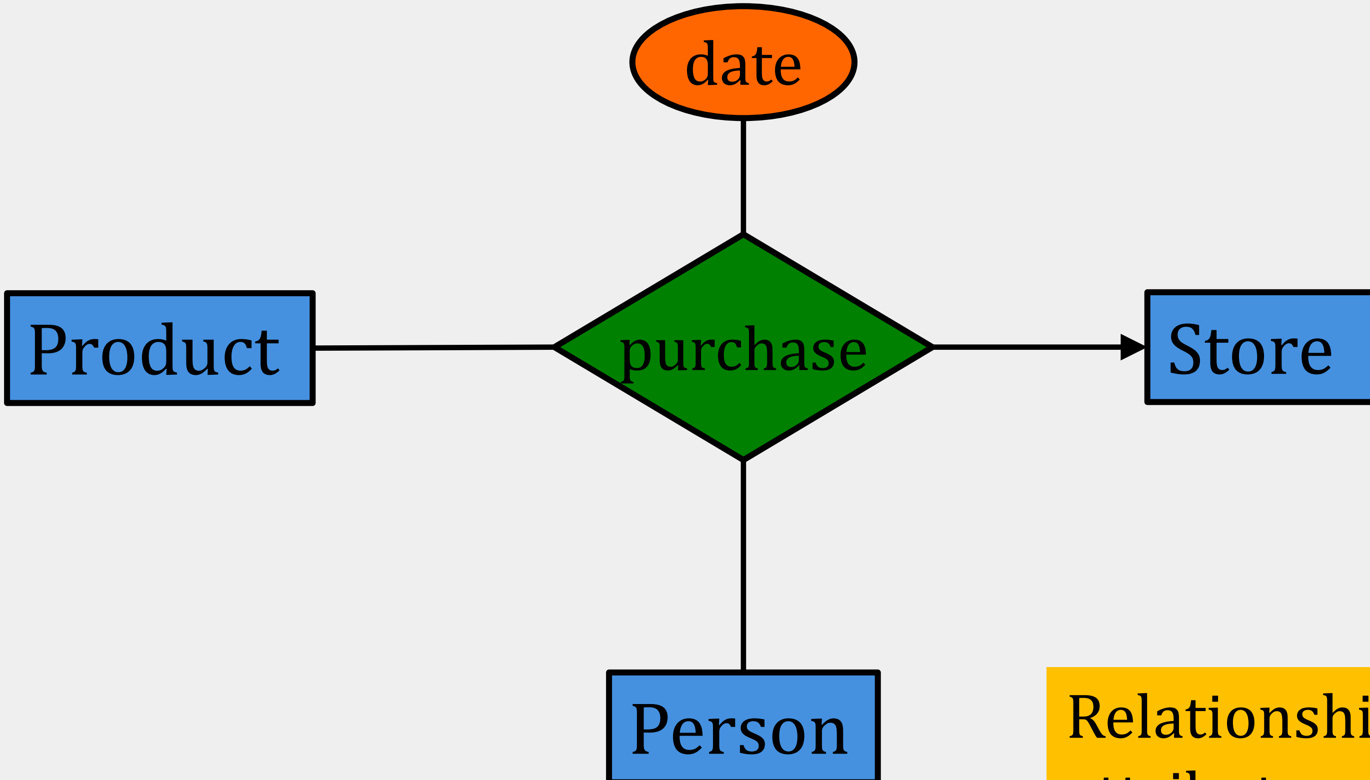


ROLES IN RELATIONSHIPS

- Label the edges to indicate the roles
- Collapse the two entity sets into one

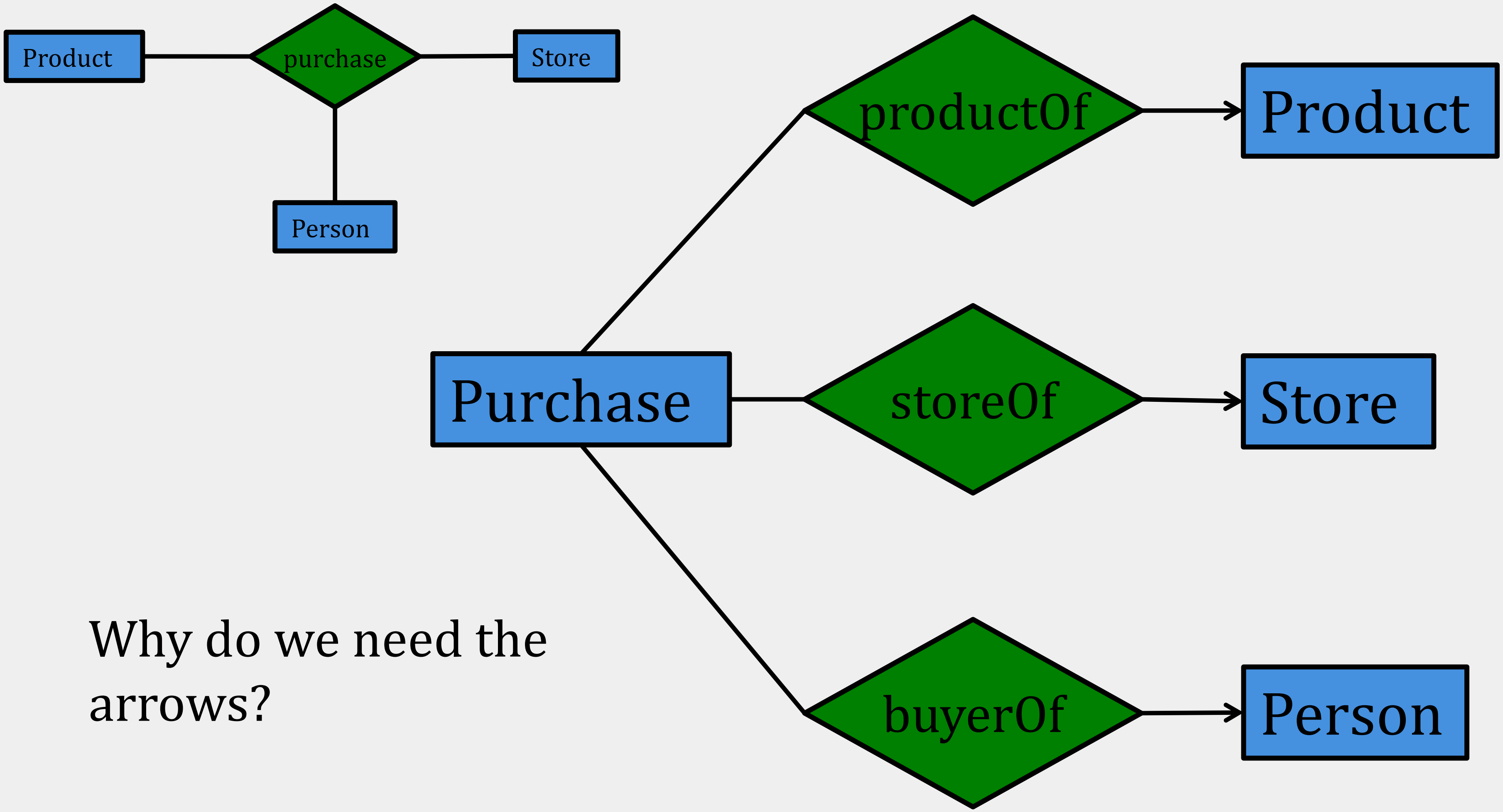


ATTRIBUTES IN RELATIONSHIPS



Relationships have attributes as well!

MULTI-WAY TO BINARY RELATIONSHIPS



Why do we need the arrows?

RELATIONSHIPS: RECAP

- Modeled as a mathematical set
- Binary and multi-way relationships
- Converting a multi-way one into many binary ones
- Constraints on the degree of the relationship
  – many-one, one-one, many-many
  – limitations of arrows
- Attributes of relationships
  – not necessary, but useful!

E/R: Additional Features

SUBCLASSES

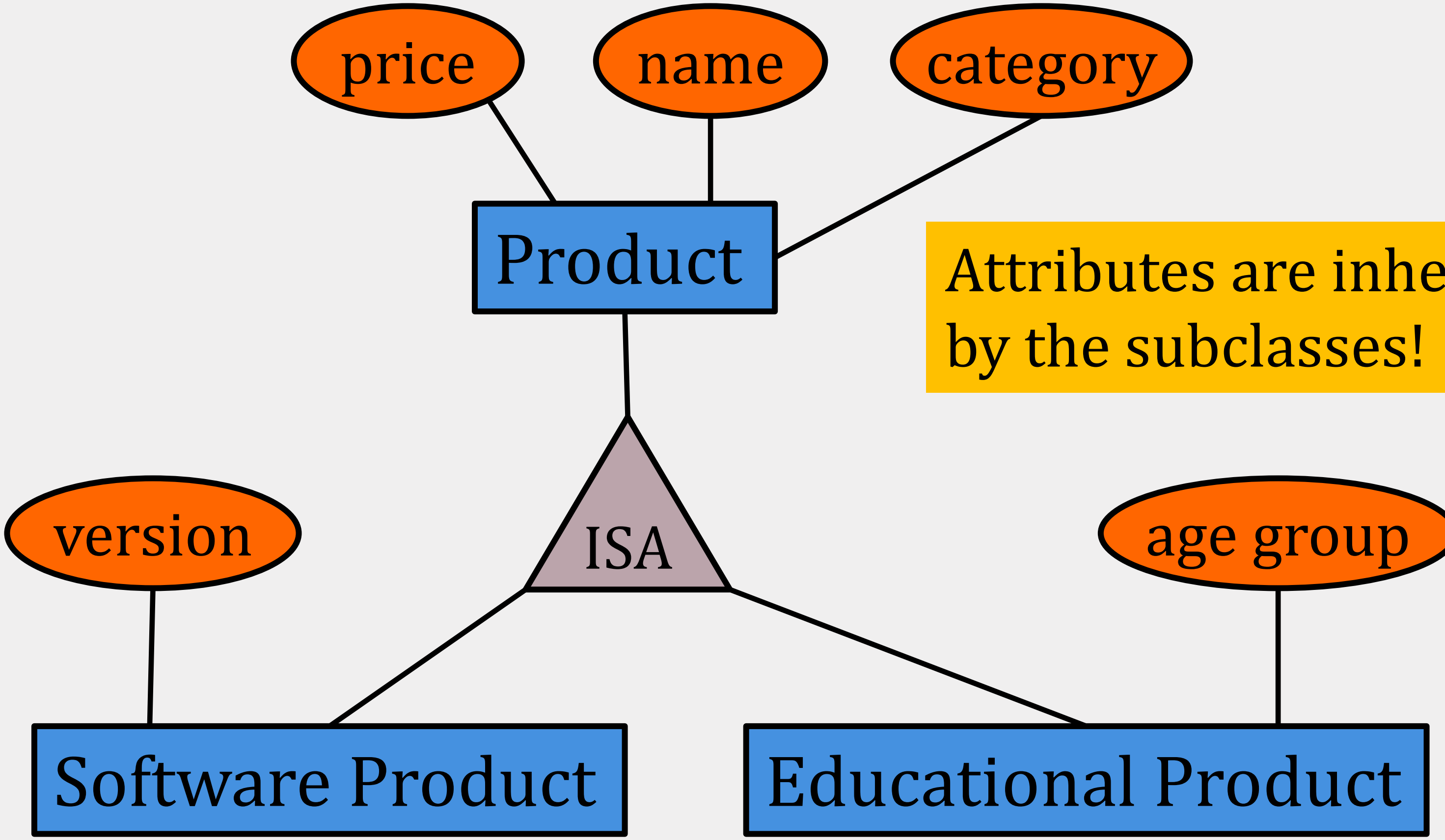
<u>subclass</u> = specialized case

= fewer entities

= more properties

- Example: Products
  - Software products
  - Educational products

SUBCLASSES



Attributes are inherited by the subclasses!

CONSTRAINTS

constraint := an assertion about the database that must be true at all times

- part of the database schema
- central in database design

When creating the ER diagram, you need to find as many constraints as possible!

TYPES OF CONSTRAINTS

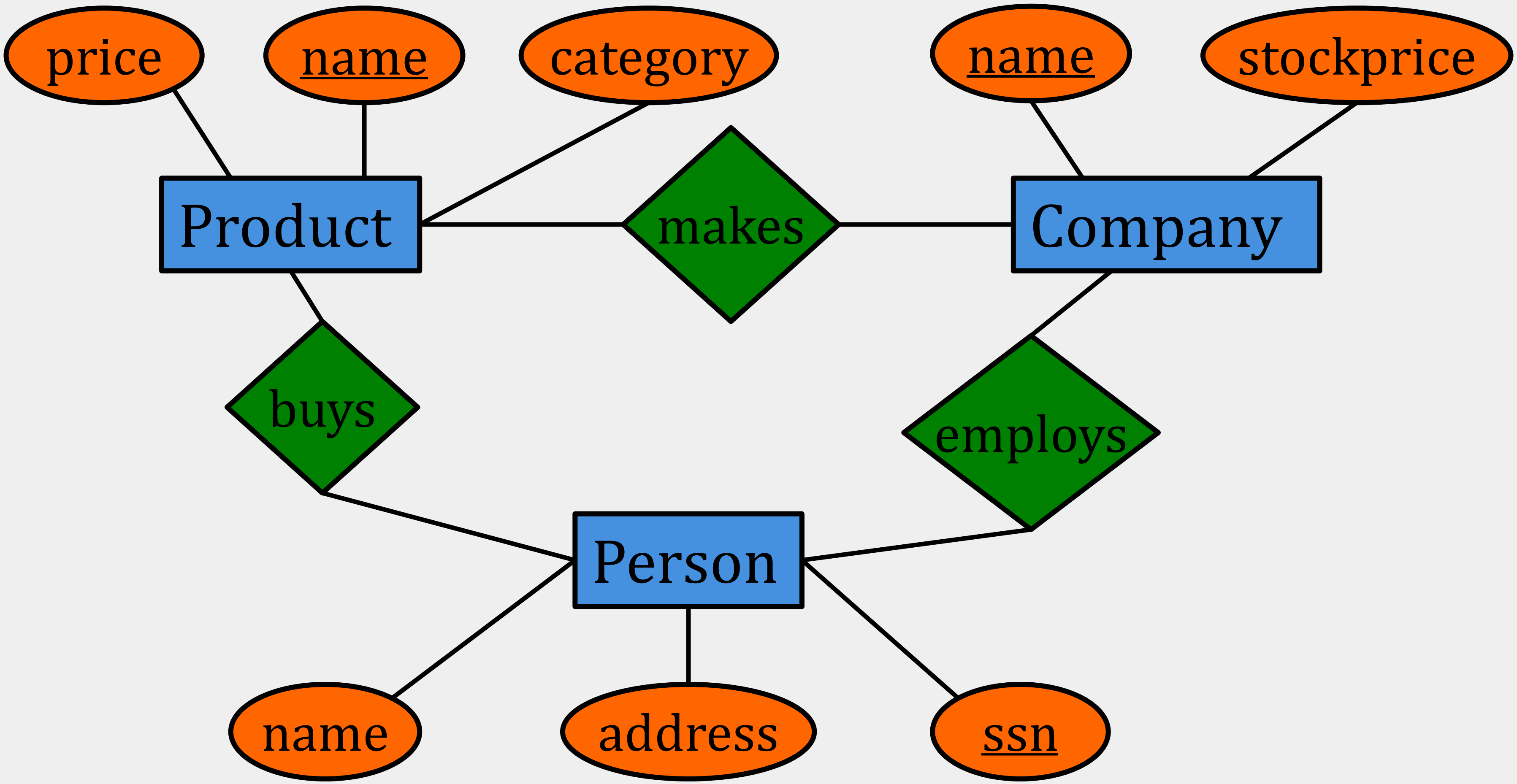
- **keys**: SSN uniquely identifies a person
- **single-value**: a person can have only one father
- **referential integrity**: if you work for a company, it must exist in the database
- **domain**: age is between 0 and 150

- **other**: e.g. at most 80 students enroll in a class

WHY DO WE NEED CONSTRAINTS?

- Give more semantics to the data
  - help us better understand it
- Prevent wrong data entry
- Allow us to refer to entities (e.g. using keys)
- Enable efficient storage and data lookup

KEY CONSTRAINTS



KEY CONSTRAINTS

Every entity set must have a key!

- A key can consist of more than one attribute
- There can be more than one key for an entity set
  - one key will be designated as primary key
- No formal way to specify multiple keys in an ER diagram

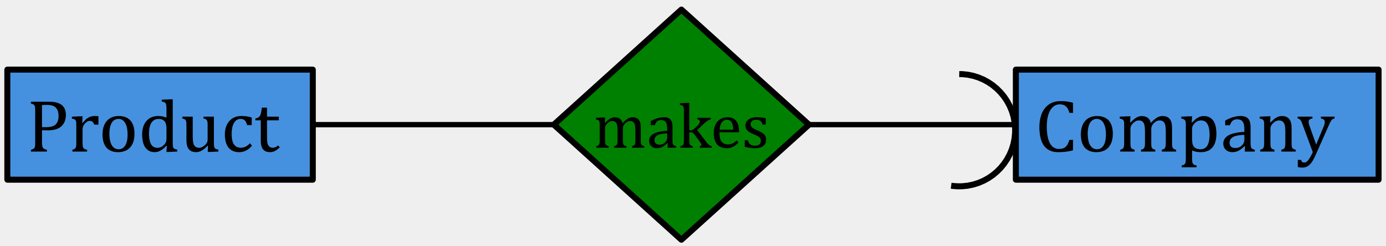
SINGLE-VALUE CONSTRAINTS

An entity may have **at most one value** for a given attribute or relationship

- an attribute of an entity set has a single value
- a many-one relation implies a single value constraint

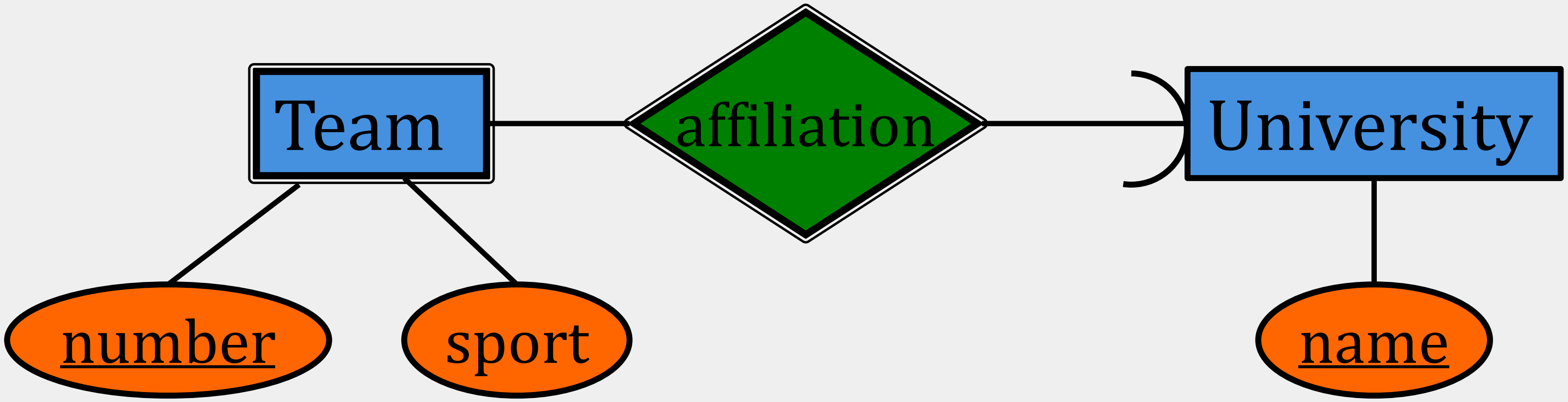
REFERENTIAL INTEGRITY CONSTRAINT

A relationship has **one value** and the value must exist



WEAK ENTITY SETS

Entity sets are **weak** when their key attributes come from other classes to which they are related

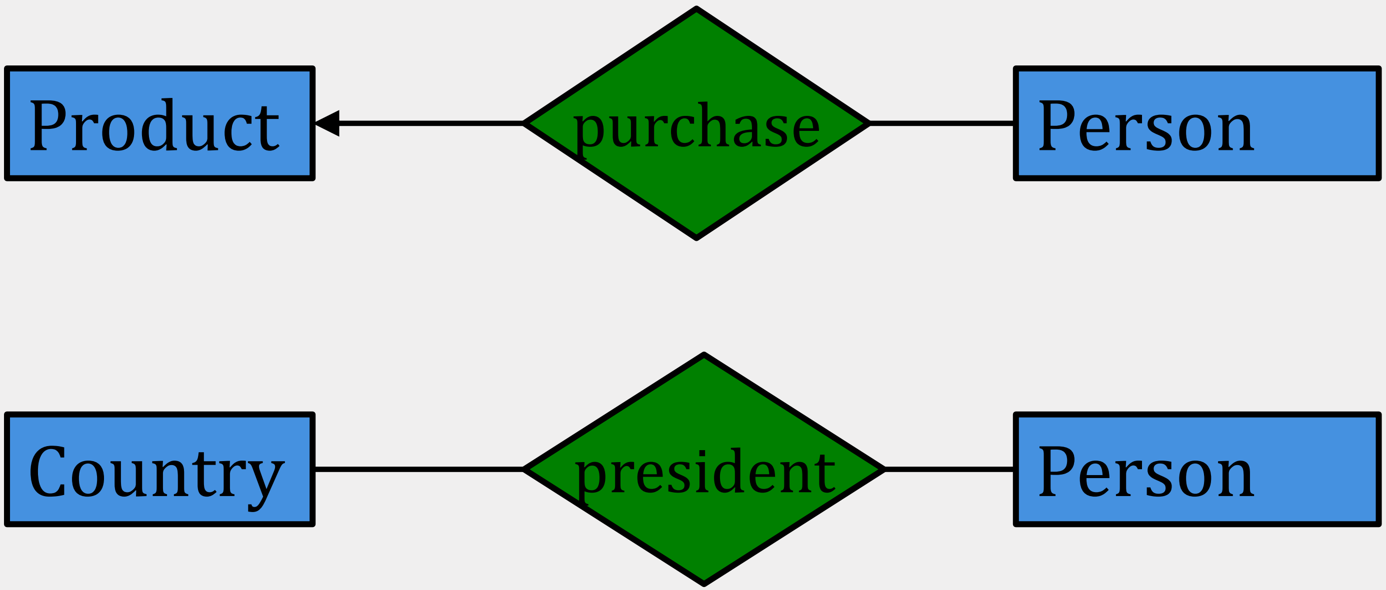


entities of an entity set need "help" to identify them uniquely!

E/R: Design Principles

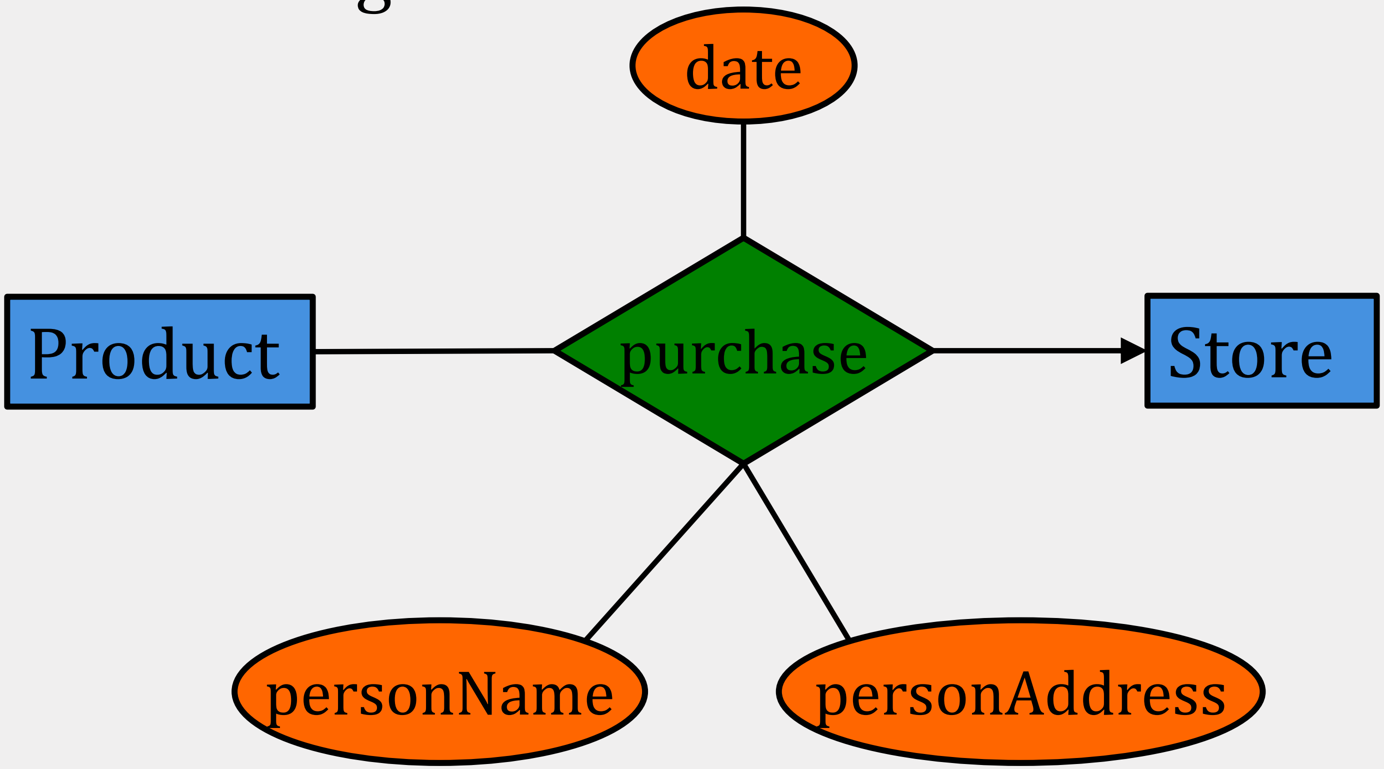
1. BE FAITHFUL TO THE APP!

What is wrong here?



2. AVOID REDUNDANCY!

What is wrong here?

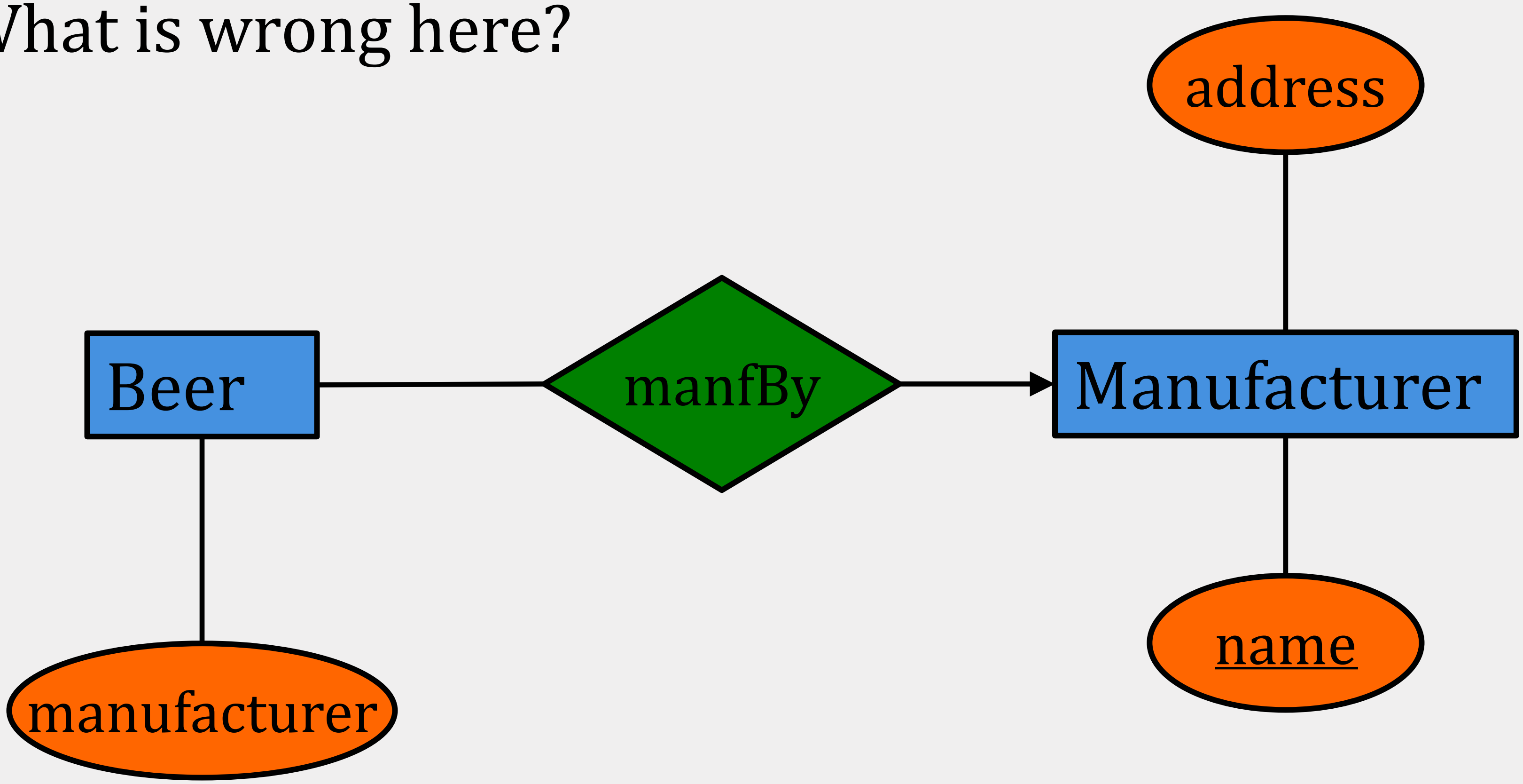


2. AVOID REDUNDANCY!

- Redundancy occurs when we say the same thing in two different ways
- Redundancy wastes space and encourages inconsistency
  - The two instances of the same fact may become inconsistent if we change one and forget to change the other

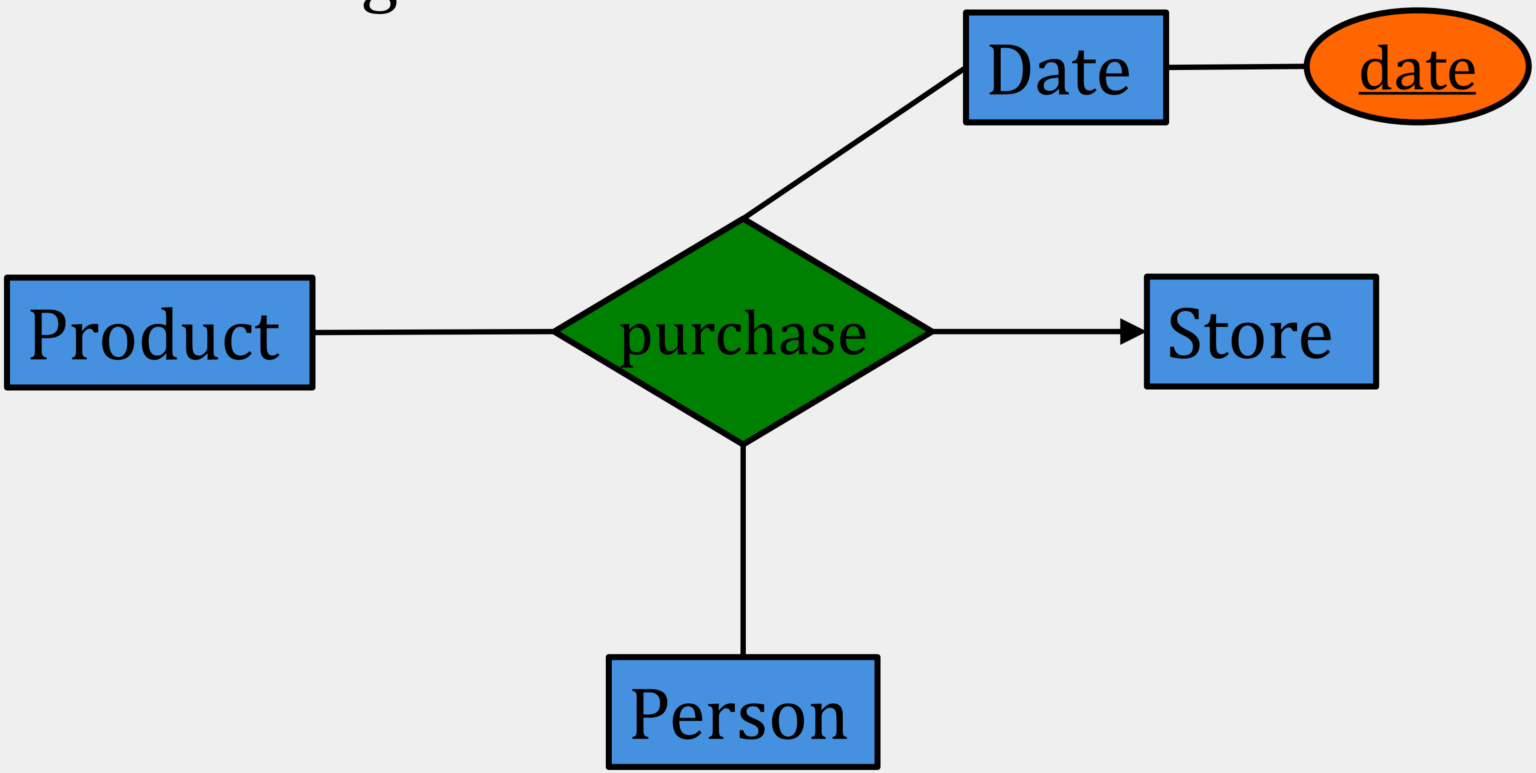
2. AVOID REDUNDANCY!

What is wrong here?



3. KEEP IT SIMPLE!

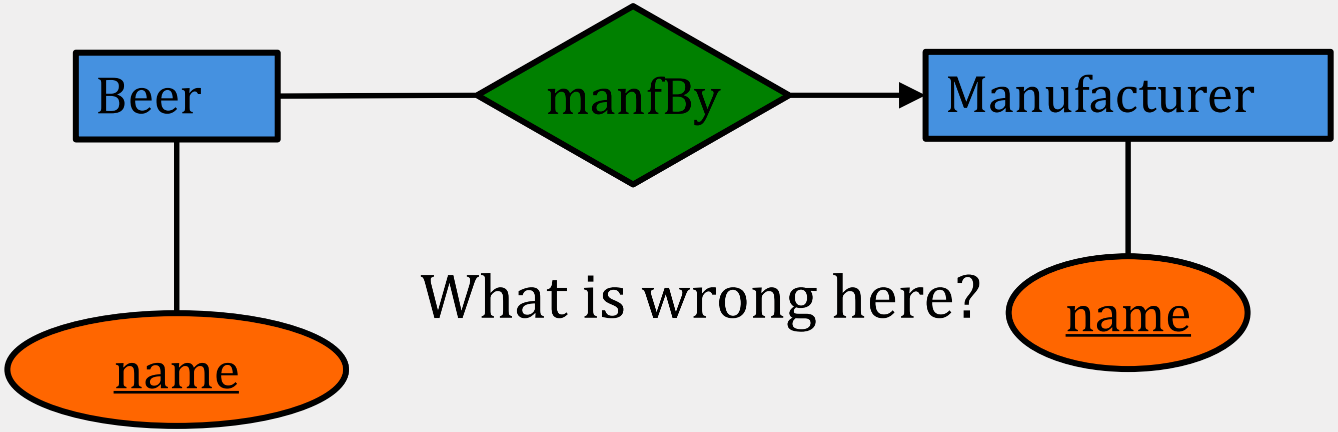
What is wrong here?



4. ATTRIBUTES OVER ENTITY SETS

An entity set should satisfy at least one of the following conditions

- it is more than the name of something; it has at least one non-key attribute
- it is the "many" in a many-one or many-many relationship



What is wrong here?

5. DON'T OVERUSE WEAK ENTITY SETS

- Beginner database designers often doubt that anything could be a key by itself
  - They make all entity sets weak, supported by all other entity sets to which they are linked
- In reality, we create unique IDs for entity sets
  - Examples: SSN, ISBN, …

E/R to Relational Model

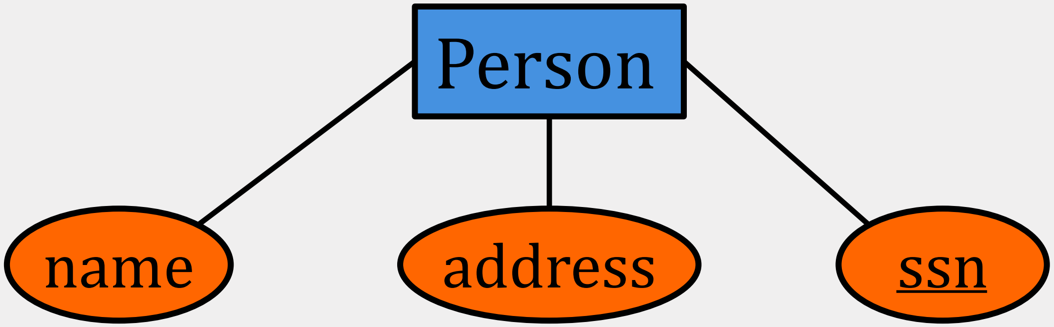
ER MODEL VS RELATIONAL MODEL

- **ER model**
  - many concepts: entities, relations, attributes, etc.
  - well-suited for capturing the app requirements
  - not well-suited for computer implementation
- **Relational model**
  - has just a single concept: relation
  - world is represented with a collection of tables
  - well-suited for efficient manipulations on computers

TRANSLATION

- Basic cases:
  - entity set **E** -- > relation with attributes of **E**
  - relationship **R** -- > relation with attributes being keys of related entity sets + attributes of **R**
- Special cases:
  - combining two relations
  - weak entity sets
  - is-a relationships

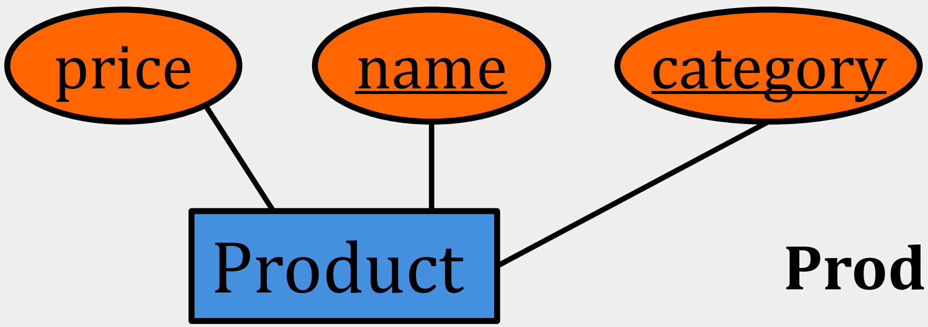
ENTITY SET TO RELATION



Person(<u>ssn</u>, name, address)

```
CREATE TABLE Person (ssn CHAR(11) PRIMARY KEY,
                     name CHAR(40),
                     address CHAR(50))
```

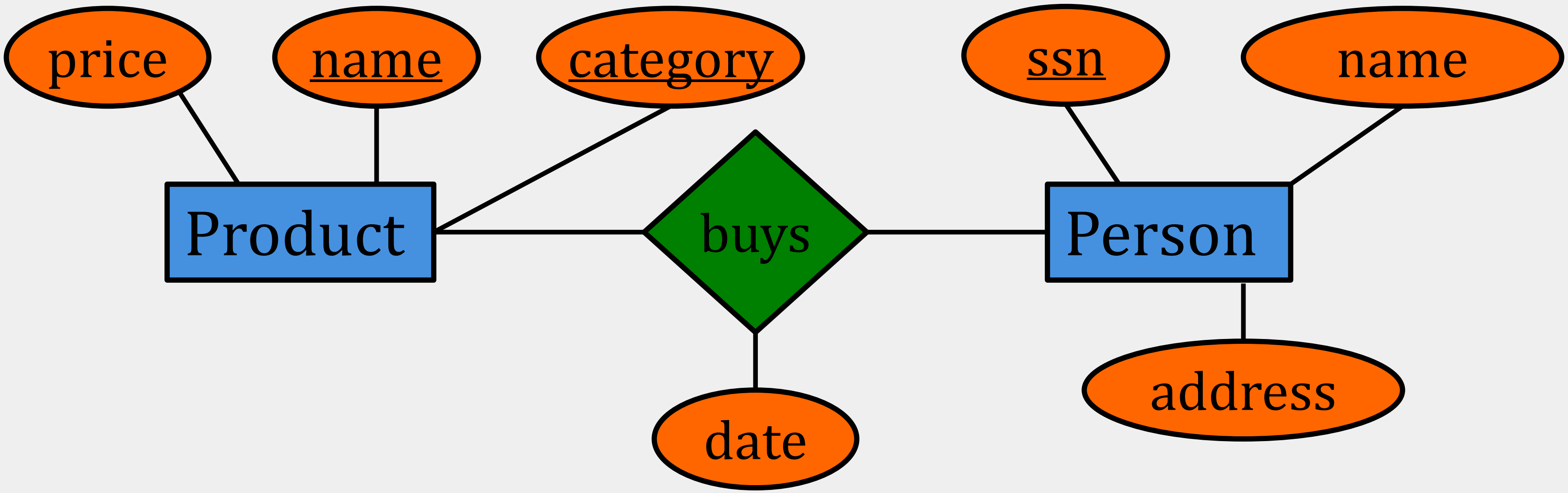
ENTITY SET TO RELATION



Product(<u>name, category</u>, price)

```
CREATE TABLE Product (name CHAR(40),
                      category CHAR(20),
                      price REAL,
                     PRIMARY KEY(name, category))
```

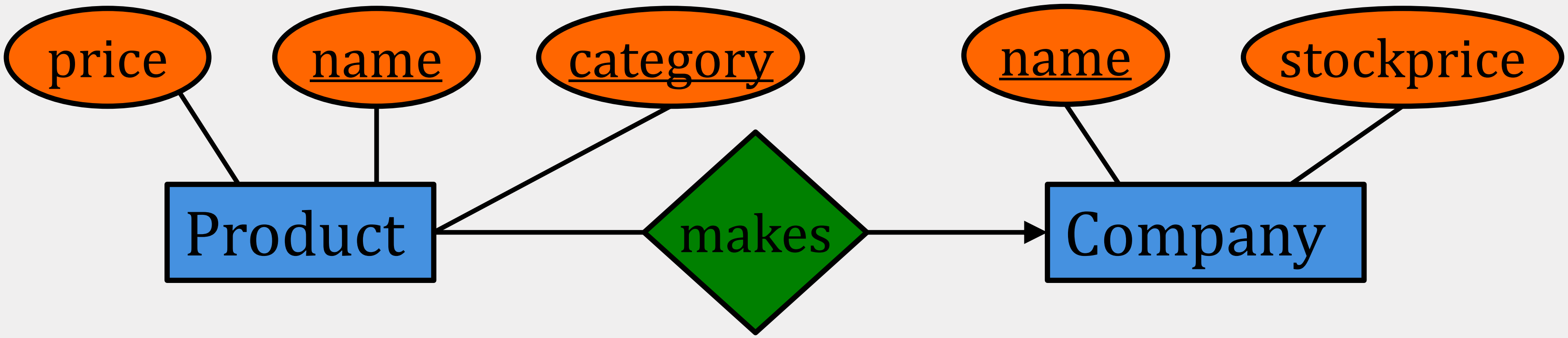
RELATIONSHIP TO RELATION



Product(name, category, price)
Person(ssn, name, address)

Buys(prodname, prodcategory, ssn, date)

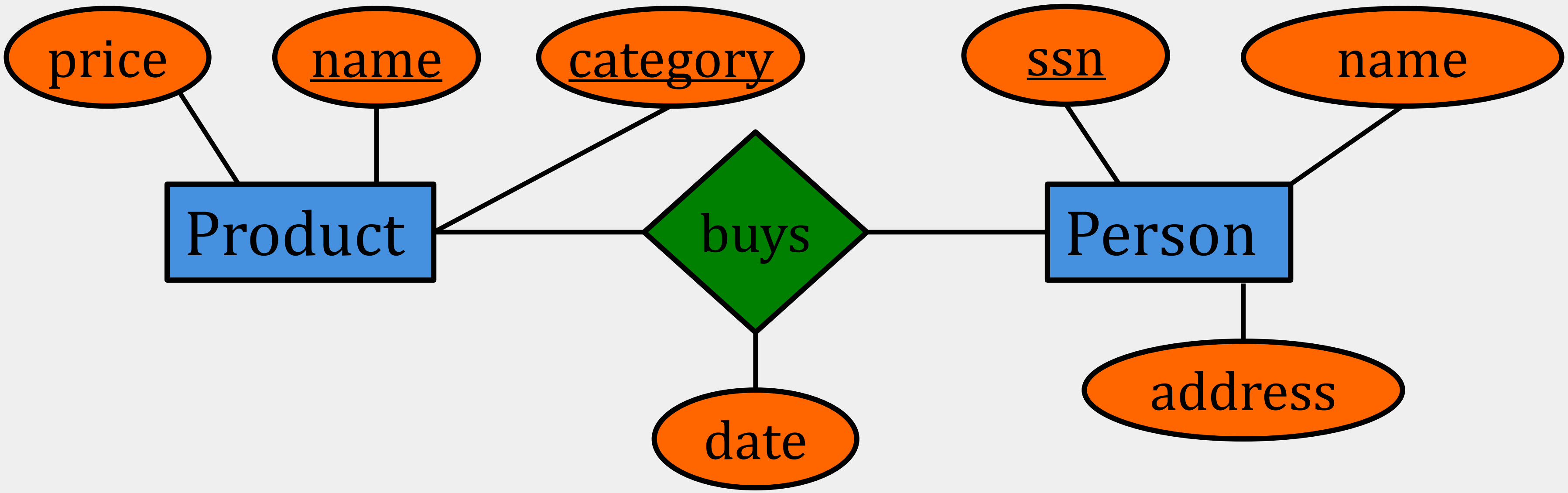
MANY-ONE RELATIONSHIPS



No need for a **Makes** relation; instead modify **Product**:

Product(<u>name, category</u>, price, company_name)
Company(<u>name</u>, stockprice)

MANY-MANY RELATIONS



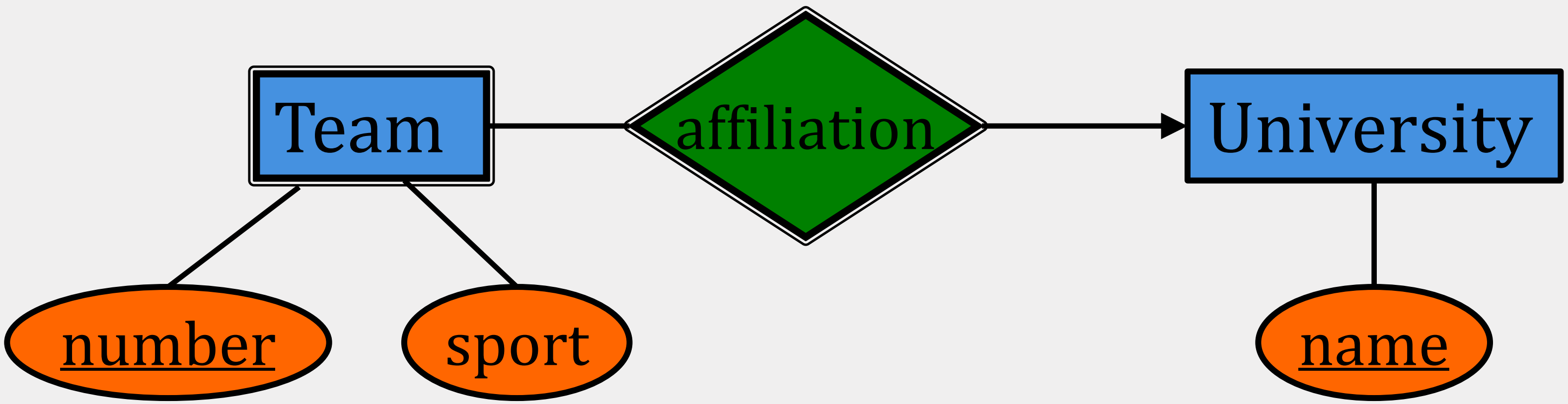
Product(<u>name, category</u>, price, ssn)

What is wrong here?

RELATIONSHIP TO RELATION: SQL

```
CREATE TABLE Buys
   (prodname CHAR(40),
    prodcategory CHAR(20),
    ssn CHAR(11),
    date DATE,
    PRIMARY KEY(prodname,prodcategory,ssn)
    FOREIGN KEY (ssn)
         REFERENCES Person,
     FOREIGN KEY (prodname, prodcategory)
         REFERENCES Product(name, category))
```

WEAK ENTITY SETS



Team(number, affiliated-university, sport)

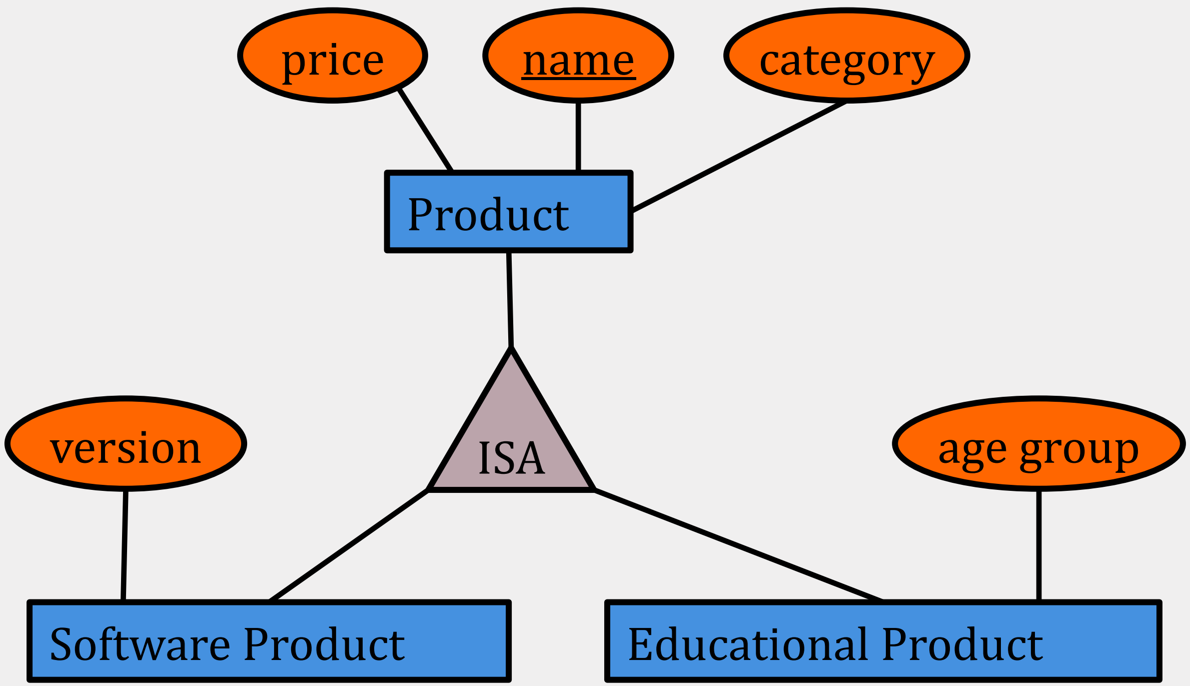
- **Affiliation** does not need a separate relation!
- Attribute '**name**' needed as part of the key

WEAK ENTITY SETS

- The relation for a weak entity set must include:
  - attributes for its complete key (including those in other entity sets)
  - its own, non-key attributes

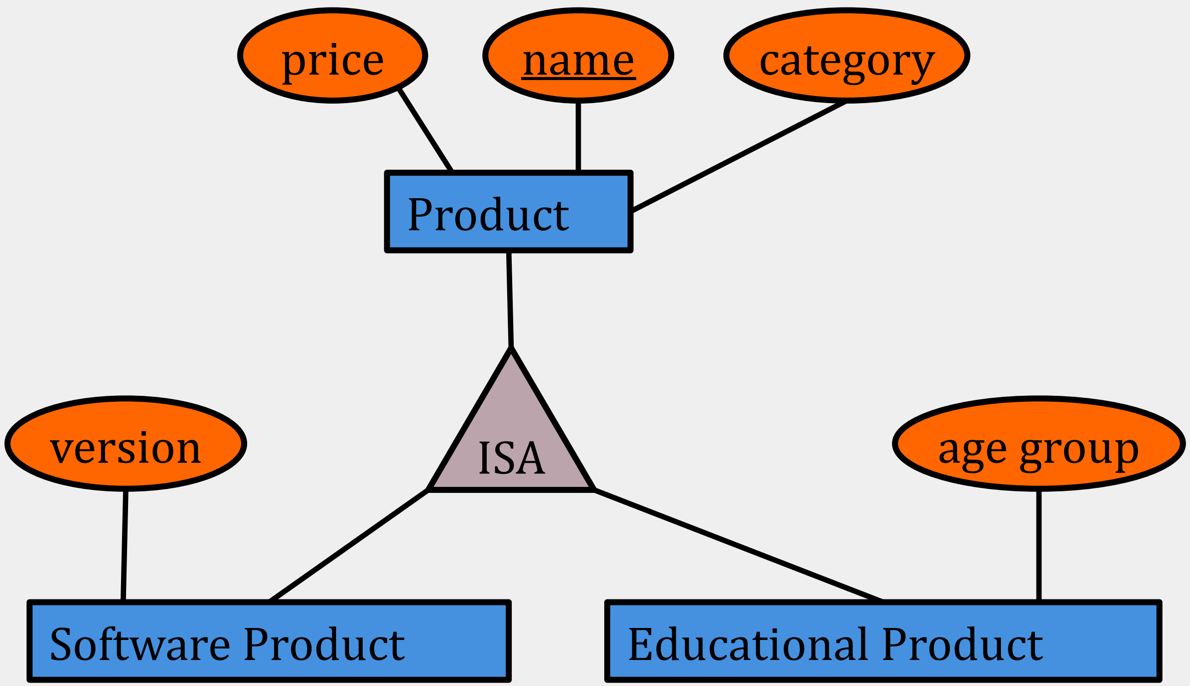
- A supporting (double-diamond) relationship is redundant and produces no relation

SUBCLASSES: OPTION 1



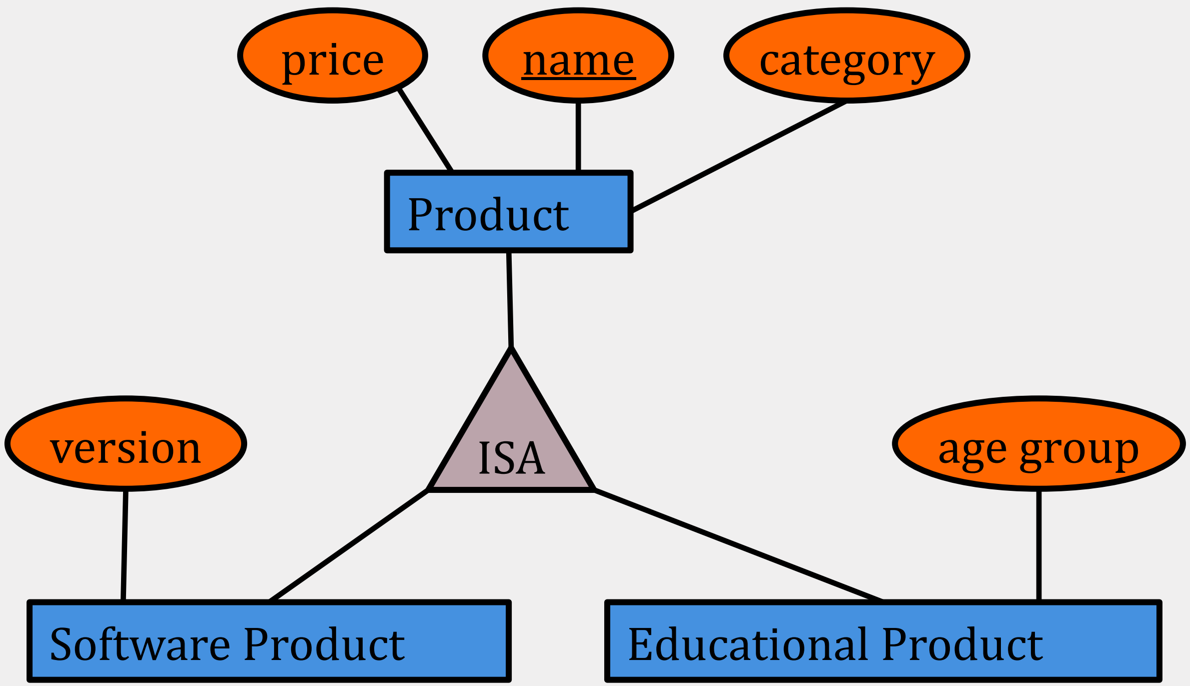
- **Product**(<u>name</u>, category, price)
- **SoftwareProduct**(<u>name</u>, category, price, version)
- **EducationalProduct**(<u>name</u>, category, price, age-group)

SUBCLASSES: OPTION 2



- **Product**(<u>name</u>, category, price)
- **SoftwareProduct**(<u>name</u>, version)
- **EducationalProduct**(<u>name</u>, age-group)

SUBCLASSES: OPTION 3



- **Product**(<u>name</u>, category, price, version, age-group)
- Use NULL to denote that the attribute makes no sense for a specific tuple

SUBCLASSES RECAP

Three approaches:

1. create a relation for each class with all its attributes
2. create one relation for each subclass with only the key attribute(s) and attributes attached to it
3. create one relation; entities have null in attributes that do not belong to them

FUNCTIONAL DEPENDENCIES

CS 564- Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

Database Design Theory:

- Functional Dependencies
- Armstrong's rules
- The Closure Algorithm
- Keys and Superkeys

HOW TO BUILD A DB APPLICATION

- Pick an application
- Figure out what to model (ER model)
  - Output: ER diagram
- Transform the ER diagram to a relational schema
- Refine the relational schema (normalization)
- Now ready to implement the schema and load the data!

DB DESIGN THEORY

- Helps us identify the “bad” schemas and improve them
  1. express constraints on the data: functional dependencies (**FDs**)
  2. use the FDs to decompose the relations

- The process, called normalization, obtains a schema in a “normal form” that guarantees certain properties
  - examples of normal forms: **BCNF**, **3NF**, …

MOTIVATING EXAMPLE

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

- What is the primary key?
  - (SSN, PhoneNumber)
- What is the problem with this schema?
  - Age and name are stored redundantly

MOTIVATING EXAMPLE

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

Problems:

- redundant storage
- update: change the age of Paris?
- insert: what if a person has no phone number?
- delete: what if Arun deletes his phone number?

SOLUTION: DECOMPOSITION

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

SSN	name	age
934729837	Paris	24
123123645	John	30
384475687	Arun	20

SSN	phoneNumber
934729837	608-374-8422
934729837	603-534-8399
123123645	608-321-1163
384475687	206-473-8221

FUNCTIONAL DEPENDENCIES

FD: DEFINITION

- **Functional dependencies** (FDs) are a form of constraint
- they generalize the concept of keys

If two tuples agree on the attributes

$$A = A_1, A_2, \dots, A_n$$

then they must agree on the attributes

$$B = B_1, B_2, \dots, B_m$$

Formally:

$$A_1, A_2, \dots, A_n \longrightarrow B_1, B_2, \dots, B_m$$

We then say that A **functionally determines** B

FD: EXAMPLE 1

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

- $SSN \longrightarrow name, age$
- $SSN, age \longrightarrow name$

FD: EXAMPLE 2

studentID	semester	courseNo	section	instructor
124434	4	CS 564	1	Paris
546364	4	CS 564	2	Arun
999492	6	CS 764	1	Anhai
183349	6	CS 784	1	Jeff

- $courseNo, section \longrightarrow instructor$
- $studentID \longrightarrow semester$

SPLITTING AN FD

- Consider the FD: $A, B \longrightarrow C, D$
- The attributes on the right are independently determined by *A, B* so we can split the FD into:
  - $A, B \longrightarrow C$ and $A, B \longrightarrow D$
- We can not do the same with attributes on the left!
  - writing $A \longrightarrow C, D$ and $B \longrightarrow C, D$ does not express the same constraint!

TRIVIAL FDS

- Not all FDs are informative:
  - $A \longrightarrow A$ holds for any relation
  - $A, B, C \longrightarrow C$ also holds for any relation

- An FD $X \longrightarrow A$ is called **trivial** if the attribute A belongs in the attribute set X
  – a trivial FD always holds!

HOW TO IDENTIFY FDS

- An FD is domain knowledge:
  - an inherent property of the application & data
  - not something we can infer from a set of tuples
- Given a table with a set of tuples
  - we can confirm that a FD **seems** to be valid
  - to infer that a FD is **definitely** invalid
  - we can **never** prove that a FD is valid

EXAMPLE 3

name	category	color	department	price
Gizmo	Gadget	Green	Toys	49
Tweaker	Gadget	Black	Toys	99
Gizmo	Stationary	Green	Office-supplies	59

Q1: Is $name \longrightarrow department$ an FD?

– not possible!

Q2: Is $name, category \longrightarrow department$ an FD ?

– we don't know!

WHY FDS?

1. keys are special cases of FDs
2. more integrity constraints for the application
3. having FDs will help us detect that a schema has redundancies and tell us how to normalize it

MORE ON FDS

- If the following FDs hold:
  - $A \longrightarrow B$
  - $B \longrightarrow C$

  then the following FD is also true:
  - $A \longrightarrow C$
- We can find more FDs like that using what we call **Armstrong's Axioms**

ARMSTRONG'S AXIOMS: 1

Reflexivity

For any subset $X \subseteq \{A_1, .., A_n\}$:

$$A_1, A_2, \ldots, A_n \longrightarrow X$$

- Examples
  - $A, B \longrightarrow B$
  - $A, B, C \longrightarrow A, B$
  - $A, B, C \longrightarrow A, B, C$

Armstrong's axioms: 2

Augmentation

For any attribute sets X, Y, Z :

if $X \longrightarrow Y$ then $X, Z \longrightarrow Y, Z$

- Examples
  - $A \longrightarrow B$ implies $A, C \longrightarrow B, C$
  - $A, B \longrightarrow C$ implies $A, B, C \longrightarrow C$

ARMSTRONG'S AXIOMS: 3

Transitivity

For any attribute sets X, Y, Z :

if $X \longrightarrow Y$ and $Y \longrightarrow Z$ then $X \longrightarrow Z$

- Examples
  - $A \longrightarrow B$ and $B \longrightarrow C$ imply $A \longrightarrow C$
  - $A \longrightarrow C, D$ and $C, D \longrightarrow E$ imply $A \longrightarrow E$

APPLYING ARMSTRONG'S AXIOMS

Product(name, category, color, department, price)

1. $name \longrightarrow color$
2. $category \longrightarrow department$
3. $color, category \longrightarrow price$

- Infer: $name, category \longrightarrow price$
    1. We apply the augmentation axiom to (1) to obtain
       (4) $name, category \longrightarrow color, category$
    2. We apply the transitivity axiom to (4), (3) to obtain
       $name, category \longrightarrow price$

APPLYING ARMSTRONG'S AXIOMS

Product(name, category, color, department, price)

1. $name \longrightarrow color$
2. $category \longrightarrow department$
3. $color, category \longrightarrow price$

- Infer: $name, category \longrightarrow color$
  1. We apply the reflexivity axiom to obtain (5) $name, category \longrightarrow name$
  2. We apply the transitivity axiom to (5), (1) to obtain $name, category \longrightarrow color$

FD CLOSURE

FD Closure
If F is a set of FDs, the closure F^+ is the set of all FDs logically implied by F

Armstrong's axioms are:

- **sound**: any FD generated by an axiom belongs in F^+
- **complete**: repeated application of the axioms will generate all FDs in F^+

CLOSURE OF ATTRIBUTE SETS

Attribute Closure

If X is an attribute set, the closure X^+ is the set of all attributes B such that:

$$X \longrightarrow B$$

In other words, X^+ includes all attributes that are functionally determined from X

EXAMPLE

Product(name, category, color, department, price)

- $name \longrightarrow color$
- $category \longrightarrow department$
- $color, category \longrightarrow price$

Attribute Closure:

- $\{name\}^+ = \{name, color\}$
- $\{name, category\}^+ =$
  $\{name, color, category, department, price\}$

THE CLOSURE ALGORITHM

- Let $X = \{A_1, A_2, \dots, A_n\}$
- **UNTIL** X doesn't change **REPEAT**:

  IF $B_1, B_2, \dots, B_m \longrightarrow C$ is an FD **AND**

  $B_1, B_2, \dots, B_m$ are all in X

  THEN add C to X

EXAMPLE

R(A, B, C, D, E, F)

- $A, B \longrightarrow C$
- $A, D \longrightarrow E$
- $B \longrightarrow D$
- $A, F \longrightarrow B$

Compute the attribute closures:

- $\{A, B\}^+ = \{A, B, C, D, E\}$
- $\{A, F\}^+ = \{A, F, B, D, E, C\}$

WHY IS CLOSURE NEEDED?

1. Does $X \longrightarrow Y$ hold?
   – we can check if Y $\subseteq X^+$
2. To compute the closure F^+ of FDs
   – for each subset of attributes X, compute X^+
   – for each subset of attributes Y $\subseteq X^+$, output the FD $X \longrightarrow Y$

KEYS & SUPERKEYS

superkey: a set of attributes $A_1, A_2, \ldots, A_n$ such that for any other attribute B in the relation:

$$A_1, A_2, \ldots, A_n \longrightarrow B$$

key (or candidate key): a minimal superkey

- none of its subsets functionally determines all attributes of the relation

If a relation has multiple keys, we specify one to be the **primary key**

COMPUTING KEYS & SUPERKEYS

- Compute X^+ for all sets of attributes X
- If $X^+ = all\ attributes$, then X is a superkey
- If no subset of X is a superkey, then X is also a key

EXAMPLE

Product(name, category, price, color)

- $name \longrightarrow color$
- $color, category \longrightarrow price$

Superkeys:

- $\{name, category\}, \{name, category, price\}$
  $\{name, category, color\}, \{name, category, price, color\}$

Keys:

- $\{name, category\}$

HOW MANY KEYS?

Q: Is it possible to have many keys in a relation **R** ?

YES!! Take relation **R**(A, B, C)with FDs

- $A, B \longrightarrow C$
- $A, C \longrightarrow B$

MINIMAL BASIS FOR FDS

- Given a set F of FDs, we know how to compute the closure F^+
- A minimal basis of F is the opposite of closure
- S is a **minimal basis** for a set F if FDs if:
  - $S^+ = F^+$
  - every FD in S has one attribute on the right side
  - if we remove any FD from S, the closure is not F^+
  - if for any FD in S we remove one or more attributes from the left side, the closure is not F^+

EXAMPLE: MINIMAL BASIS

Example:

- $A \longrightarrow B$
- $A, B, C, D \longrightarrow E$
- $E, F \longrightarrow G, H$
- $A, C, D, F \longrightarrow E, G$

STEP 1: SPLIT THE RIGHT HAND SIDE

- $A \longrightarrow B$
- $A, B, C, D \longrightarrow E$
- $E, F \longrightarrow G$
- $E, F \longrightarrow H$
- $A, C, D, F \longrightarrow E$
- $A, C, D, F \longrightarrow G$

STEP 2: REMOVE REDUNDANT FDS

- $A \longrightarrow B$
- $A, B, C, D \longrightarrow E$
- $E, F \longrightarrow G$
- $E, F \longrightarrow H$
- ~~$A, C, D, F \longrightarrow E$~~
- ~~$A, C, D, F \longrightarrow G$~~

can be removed, since these FDs are logically implied by the remaining FDs

STEP 3: CLEAN UP THE LEFT HAND SIDE

- $A \longrightarrow B$
- $A, \cancel{B}, C, D \longrightarrow E$
- $E, F \longrightarrow G$
- $E, F \longrightarrow H$

B can be safely removed because of the first FD

EXAMPLE: FINAL RESULT

- $A \longrightarrow B$
- $A, C, D \longrightarrow E$
- $E, F \longrightarrow G$
- $E, F \longrightarrow H$

RECAP

- FDs and (super)keys
- Reasoning with FDs:
  - given a set of FDs, infer all implied FDs
  - given a set of attributes X, infer all attributes that are functionally determined by X
- Next we will look at how to use them to detect that a table is "bad"

DECOMPOSITION & SCHEMA NORMALIZATION

CS 564 - Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

- Bad schemas lead to redundancy
- To "correct" bad schemas: decompose relations
  - lossless-join
  - dependency preserving
- Desired normal forms
  - **BCNF**
  - **3NF**

DB DESIGN THEORY

- Helps us identify the “bad” schemas and improve them
  1. express constraints on the data: functional dependencies (**FDs**)
  2. use the FDs to decompose the relations

- The process, called normalization, obtains a schema in a “normal form” that guarantees certain properties
  – examples of normal forms: **BCNF**, **3NF**, …

SCHEMA DECOMPOSITION

WHAT IS A DECOMPOSITION?

We decompose a relation **R**(A_1, …, A_n) by creating

- **R₁**(B_1, .., B_m)
- **R₂**(C_1,…, C_k)

where $\{B_1, \dots, B_m\} \cup \{C_1, \dots, C_k\} = \{A_1, \dots A_n\}$

- The instance of $\mathbf{R_1}$ is the projection of **R** onto B_1, .., B_m
- The instance of $\mathbf{R_2}$ is the projection of **R** onto C_1, .., C_l

In general we can decompose a relation into multiple relations.

EXAMPLE: DECOMPOSITION

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

SSN	name	age
934729837	Paris	24
123123645	John	30
384475687	Arun	20

SSN	phoneNumber
934729837	608-374-8422
934729837	603-534-8399
123123645	608-321-1163
384475687	206-473-8221

DECOMPOSITION DESIDERATA

What should a good decomposition achieve?

1. minimize redundancy
2. avoid information loss (lossless-join)
3. preserve the FDs (dependency preserving)
4. ensure good query performance

EXAMPLE: INFORMATION LOSS

name	age	phoneNumber
Paris	24	608-374-8422
John	24	608-321-1163
Arun	20	206-473-8221

Decompose into:

R_1(name, age)

R_2(age, phoneNumber)

name	age
Paris	24
John	24
Arun	20

age	phoneNumber
24	608-374-8422
24	608-321-1163
20	206-473-8221

We can't figure out which phoneNumber corresponds to which person!

LOSSLESS-JOIN DECOMPOSITION

$$
\begin{tikzcd}
 & \mathbf{R}(A, B, C) \arrow[dl] \arrow[dr] & \\
\mathbf{R_1}(A, B) \arrow[dr] & & \mathbf{R_2}(B, C) \arrow[dl] \\
 & \mathbf{R'}(A, B, C) &
\end{tikzcd}
$$

decompose (projection)

recover (natural join)

A natural join is a join on the same attribute names

A schema decomposition is **lossless-join** if for any initial instance **R**, **R = R'**

THE CHASE ALGORITHM

- The chase algorithm is a classic database technique that can be used to check for lossless-join decomposition.

Running example

- relation **R**(A, B, C, D, E)
- FDs: $A \longrightarrow B, C \quad D \longrightarrow E$

Question: is the following decomposition lossless-join?

$\mathbf{R_1}$(A, D) $\mathbf{R_2}$(A, B, C) $\mathbf{R_3}$(D, E)

CHASE: INITIALIZATION

- We create a table with the attributes of the original relation
- We add one row for each relation we split to

	A	B	C	D	E
$\mathbf{R_1}$(A, D)	a	b_1	c_1	d	e_1
$\mathbf{R_2}$(A, B, C)	a	b	c	d_2	e_2
$\mathbf{R_3}$(D, E)	a_3	b_3	c_3	d	e

Attribute not in the relation gets a subscript

Attribute in the relation - no subscript

CHASE: MAIN ALGORITHM

- We proceed in iterations
- At every iteration, we check whether an FD is violated, and if so, we "force" it to hold by removing subscripts

	A	B	C	D	E
R₁(A, D)	a	b_1	c_1	d	$e_1 \rightarrow$ e
R₂(A, B, C)	a	b	c	d_2	e_2
R₃(D, E)	a_3	b_3	c_3	d	e

$A \longrightarrow B, C$
$D \longrightarrow E$

The FD $D \longrightarrow E$ is violated, so we need to drop the subscript from the first row

CHASE: MAIN ALGORITHM

$A \longrightarrow B, C$
$D \longrightarrow E$

A	B	C	D	E
a	b_1	c_1	d	e_1
a	b	c	d_2	e_2
a_3	b_3	c_3	d	e

$D \longrightarrow E$

A	B	C	D	E
a	b_1	c_1	d	e
a	b	c	d_2	e_2
a_3	b_3	c_3	d	e

$A \longrightarrow B, C$

A	B	C	D	E
a	b	c	d	e
a	b	c	d_2	e_2
a_3	b_3	c_3	d	e

At the end of the chase:
- If there is a row w/o subscripts, we can say that the decomposition is lossless-join
- otherwise, we output no

MORE EXAMPLES

- relation **R**(A, B, C, D)
- FD $A \longrightarrow B, C$

Lossless-join

- decomposition into $\mathbf{R_1}$(A, B, C) and $\mathbf{R_2}$(A, D)

Not lossless-join

- decomposition into $\mathbf{R_1}$(A, B, C) and $\mathbf{R_2}$(D)

DEPENDENCY PRESERVING

Given **R** and a set of FDs F, we decompose **R** into $\mathbf{R_1}$ and $\mathbf{R_2}$. Suppose:

- $\mathbf{R_1}$ has a set of FDs F_1
- $\mathbf{R_2}$ has a set of FDs F_2
- F_1 and F_2 are computed from F

A decomposition is **dependency preserving** if by enforcing F_1 over $\mathbf{R_1}$ and F_2 over $\mathbf{R_2}$, we can enforce F over **R**

A NOTE ON FDS OF SPLIT RELATIONS

Given **R** and a set of FDs F, we decompose **R** into $\mathbf{R_1}$ and $\mathbf{R_2}$. How do we find the FDs F_1 that hold for $\mathbf{R_1}$?

- It is not enough to only keep the FDs from F with attributes in $\mathbf{R_1}$
- Instead, we need to find the non-trivial FDs in the fd closure of F with attributes in $\mathbf{R_1}$

Example: **R**(A, B, C) with FDs: $A \longrightarrow B \quad B \longrightarrow C$

- For $\mathbf{R_1}$(A, C) $\quad F_1 = A \longrightarrow C$

GOOD EXAMPLE

Person(SSN, name, age, canDrink)

- $SSN \longrightarrow name, age$
- $age \longrightarrow canDrink$

decomposes into

- $\mathbf{R_1}$(SSN, name, age)
  - $SSN \longrightarrow name, age$
- $\mathbf{R_2}$(age, canDrink)
  - $age \longrightarrow canDrink$

BAD EXAMPLE

R(A, B, C)

- $A \longrightarrow B$
- $B, C \longrightarrow A$

Decomposes into:

- **R_1**(A, B)
  - $A \longrightarrow B$
- **R_2**(A, C)
  - no FDs here!!

R_1

A	B
a_1	b
a_2	b

R_2

A	C
a_1	c
a_2	c

recover

A	B	C
a_1	b	c
a_2	b	c

The recovered table violates $B, C \longrightarrow A$

NORMAL FORMS

A **normal form** represents a “good” schema design:

- 1NF (flat tables/atomic values)
- 2NF
- **3NF**
- **BCNF**
- 4NF
- …

more
restrictive

BCNF Decomposition

BOYCE-CODD NORMAL FORM (BCNF)

A relation **R** is in **BCNF** if whenever $X \longrightarrow B$ is a non-trivial FD, then X is a superkey in **R**

Equivalent definition: for every attribute set X

- either $X^+ = X$
- or $X^+ = \mathit{all\ attributes}$

BCNF EXAMPLE 1

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

$$SSN \longrightarrow name, age$$

- **key** = $\{SSN, phoneNumber\}$
- $SSN \longrightarrow name, age$ is a “bad” FD
- The above relation is **not** in BCNF!

BCNF Example 2

SSN	**name**	**age**
934729837	Paris	24
123123645	John	30
384475687	Arun	20

$$SSN \longrightarrow name, age$$

- **key** = $\{SSN\}$
- The above relation is in BCNF!

BCNF Example 3

SSN	phoneNumber
934729837	608-374-8422
934729837	603-534-8399
123123645	608-321-1163
384475687	206-473-8221

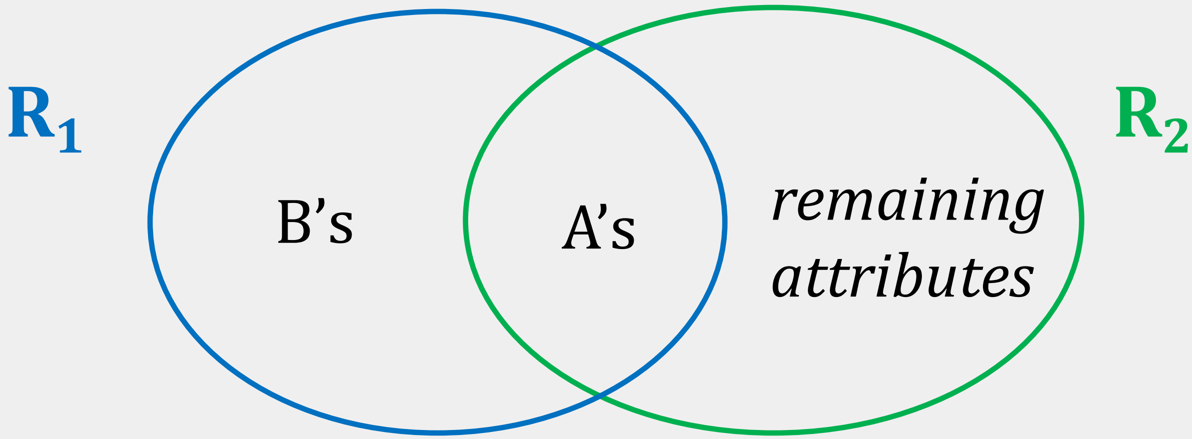
- **key** = $\{SSN, phoneNumber\}$
- The above relation is in BCNF!
- Is it possible that a binary relation is not in BCNF?

BCNF DECOMPOSITION

- Find an FD that violates the BCNF condition

$$A_1, A_2, \dots, A_n \longrightarrow B_1, B_2, \dots, B_m$$

- Decompose **R** to **R_1** and **R_2**:

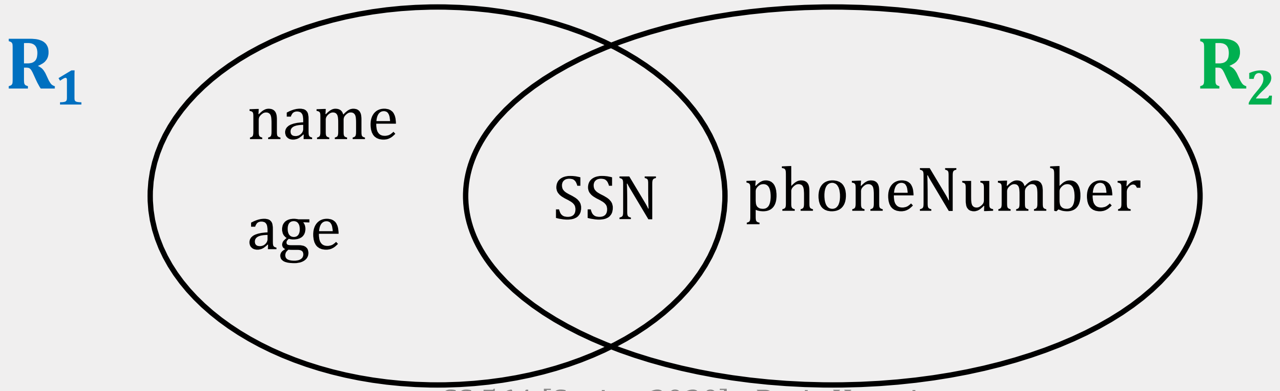


- Continue until no BCNF violations are left

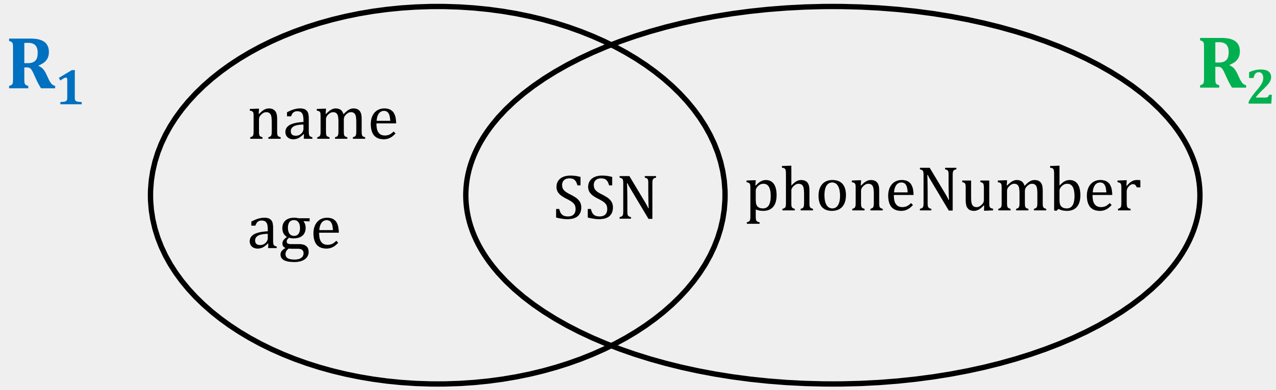
EXAMPLE

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

- The FD $SSN \longrightarrow name, age$ violates BCNF
- Split into two relations $\mathbf{R_1}$, $\mathbf{R_2}$ as follows:



EXAMPLE CONT'D



$$SSN \longrightarrow name, age$$

SSN	name	age
934729837	Paris	24
123123645	John	30
384475687	Arun	20

SSN	phoneNumber
934729837	608-374-8422
934729837	603-534-8399
123123645	608-321-1163
384475687	206-473-8221

BCNF DECOMPOSITION PROPERTIES

The BCNF decomposition:

– removes certain types of redundancy

– is lossless-join

– is not always dependency preserving

BCNF IS LOSSLESS-JOIN

Example:

R(A, B, C) with $A \longrightarrow B$ decomposes into:

$\mathbf{R_1}$(A, B) and $\mathbf{R_2}$(A, C)

- The BCNF decomposition always satisfies the lossless-join criterion!

BCNF IS NOT DEPENDENCY PRESERVING

R(A, B, C)

- $A \longrightarrow B$
- $B, C \longrightarrow A$

There may not exist any BCNF decomposition that is FD preserving!

The BCNF decomposition is:

- $\mathbf{R_1}$(A, B) with FD $A \longrightarrow B$
- $\mathbf{R_2}$(A, C) with no FDs

BCNF EXAMPLE (1)

Books (author, gender, booktitle, genre, price)

- $author \longrightarrow gender$
- $booktitle \longrightarrow genre, price$

What is the candidate key?

- *(author, booktitle)* is the only one!

Is is in BCNF?

- **No**, because the left hand side of both (not trivial) FDs is not a superkey!

BCNF EXAMPLE (2)

Books (author, gender, booktitle, genre, price)

- $author \longrightarrow gender$
- $booktitle \longrightarrow genre, price$

Splitting **Books** using the FD $author \longrightarrow gender$:

- **Author** (author, gender)
  FD: $author \longrightarrow gender$ in BCNF!
- **Books2** (authos, booktitle, genre, price)
  FD: $booktitle \longrightarrow genre, price$ not in BCNF!

BCNF EXAMPLE (3)

Books (author, gender, booktitle, genre, price)

- $author \longrightarrow gender$
- $booktitle \longrightarrow genre, price$

Splitting **Books** using the FD $author \longrightarrow gender$:

- **Author** (author, gender)
  FD: $author \longrightarrow gender$ in BCNF!

- Splitting **Books2** (author, booktitle, genre, price):
  - **BookInfo** (booktitle, genre, price)
    FD: $booktitle \longrightarrow genre, price$ in BCNF!
  - **BookAuthor** (author, booktitle) in BCNF!

Third Normal Form (3NF)

3NF DEFINITION

A relation **R** is in **3NF** if whenever $X \longrightarrow A$, one of the following is true:

- $A \in X$ (trivial FD)
- X is a superkey
- A is part of some key of **R** (prime attribute)

BCNF implies 3NF !!

3NF CONT'D

- Example: **R**(A, B, C) with $A, B \longrightarrow C$ and $C \longrightarrow A$
  - is in 3NF. Why?
  - is not in BCNF. Why?

- Compromise used when BCNF not achievable: *aim for BCNF and settle for 3NF*
- Lossless-join and dependency preserving decomposition into a collection of 3NF relations is always possible!

3NF ALGORITHM

1. Apply the algorithm for BCNF decomposition until all relations are in 3NF (we can stop earlier than BCNF)
2. Compute a minimal basis F' of F
3. For each non-preserved FD $X \longrightarrow A$ in F', add a new relation R(X, A)

3NF EXAMPLE (1)

Start with relation **R** (A, B, C, D) with FDs:

- $A \longrightarrow D$
- $A, B \longrightarrow C$
- $A, D \longrightarrow C$
- $B \longrightarrow C$
- $D \longrightarrow A, B$

Step 1: find a BCNF decomposition

- **R1** (B, C)
- **R2** (A, B, D)

3NF EXAMPLE (2)

Start with relation **R** (A, B, C, D) with FDs:

- $A \longrightarrow D$
- $A, B \longrightarrow C$
- $A, D \longrightarrow C$
- $B \longrightarrow C$
- $D \longrightarrow A, B$

Step 2: compute a minimal basis of the original set of FDs:

- $A \longrightarrow D$
- $B \longrightarrow C$
- $D \longrightarrow A$
- $D \longrightarrow B$

3NF EXAMPLE (3)

Start with relation **R** (A, B, C, D) with FDs:

- $A \longrightarrow D$
- $A, B \longrightarrow C$
- $A, D \longrightarrow C$
- $B \longrightarrow C$
- $D \longrightarrow A, B$

Step 3: add a new relation for any FD in the basis that is not satisfied:

- all the dependencies in *F'* are satisfied!
- the resulting decomposition **R1**, **R2** is also BCNF!

IS NORMALIZATION ALWAYS GOOD?

- Example: suppose A and B are always used together, but normalization says they should be in different tables
  - decomposition might produce unacceptable performance loss
- Example: data warehouses
  - huge historical DBs, rarely updated after creation
  - joins expensive or impractical

Relational Algebra

CS 564- Fall 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

- Relational Algebra
  - query language for relations
- Basic Operations
  - selection, projection
  - difference, union
  - cross-product, renaming
- Derived Operations
  - join, natural join, equi-join, division, etc

RELATIONAL QUERY LANGUAGES

- allow the *manipulation* and *retrieval* of data from a database
- two types of query languages:
  - Declarative: describe what a user wants, rather than how to compute it
    - Tuple Relational Calculus (**TRC**)
    - Domain Relational Calculus (**DRC**)
  - Procedural: operational, useful for representing execution plans
    - Relational Algebra (**RA**)

WHAT IS RELATIONAL ALGEBRA?

- **algebra**: mathematical system consisting of
  - operands: variables or values from which new values can be constructed
  - operators: symbols denoting procedures that construct new values from given values
- **relational algebra**: an algebra whose operands are relations or variables that represent relations
  - operators do the most common things that we need to do with relations in a database
  - can be used as a query language for relations

RELATIONAL ALGEBRA: PRELIM

- **Query**:
  - Input: relational instances
  - Output: relational instances
  - specified using the schemas
    - may produce different results for different instances
    - the schema of the result is fixed
- there are two types of notation for attributes:
  - positional (e.g. 2, 4)
  - named-field (e.g. C.name, Person.SSN)

RELATIONAL ALGEBRA: PRELIM

- Basic operations:
  - *Selection* $\{\sigma\}$: selects a subset of rows
  - *Projection* $\{\pi\}$: deletes columns
  - *Cross-product* $\{\times\}$: combines two relations
  - *Set-difference* $\{-\}$
  - *Union* $\{\cup\}$
- When the relations have named fields:
  - *Renaming* $\{\rho\}$
- Additional operations:
  - *Intersection, join, division*

KEEP IN MIND!

- SQL uses multisets, however in Relational Algebra we will consider relations as **sets**
- We will consider the **named perspective**, where every attribute must have a unique name

The attribute order in a relation does not matter!

BASIC OPERATIONS

SELECTION

Notation: $\sigma_C(R)$

- C is a condition that refers to the attributes of R
- outputs the rows of R that satisfy C
- output schema: same as input schema

Example

- $\sigma_{age>24}(Person)$
- $\sigma_{age>24\ and\ age\leq 28}(Person)$
- $\sigma_{age>24\ and\ name=\text{"}Paris\text{"}}(Person)$

```
SELECT *
FROM Person
WHERE age > 24 ;
```

SELECTION: EXAMPLE

Person

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	25	206-473-8221

$$\downarrow \; \sigma_{age>24}(Person)$$

SSN	name	age	phoneNumber
123123645	John	30	608-321-1163
384475687	Arun	25	206-473-8221

PROJECTION

Notation: $\pi_{A_1,A_2,\ldots,A_n}(R)$

- outputs only the columns $A_1, A_2, \ldots, A_n$
- removes any duplicate tuples
- output schema: $R(A_1, A_2, \ldots, A_n)$

Example

- $\pi_{SSN,age}(Person)$
- $\pi_{SSN,phoneNumber,age}(Person)$

```sql
SELECT DISTINCT SSN,age
FROM Person ;
```

PROJECTION: EXAMPLE

Person

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

$$\pi_{SSN,name}(Person)$$

SSN	name
934729837	Paris
123123645	John
384475687	Arun

RA OPERATORS ARE COMPOSITIONAL

```sql
SELECT DISTINCT SSN,age
FROM Person
WHERE age > 24 ;
```

Two logically equivalent expressions in RA:

- $\pi_{SSN,age}\left(\sigma_{age>24}(Person)\right)$
- $\sigma_{age>24}(\pi_{SSN,age}(Person))$

UNION

Notation: $R_1 \cup R_2$

- outputs all tuples in R_1 or R_2
- both relations must have the same schema!
- output schema: same as input

A	B
a_1	b_1
a_2	b_1
a_2	b_2

$\cup$

A	B
a_1	b_1
a_3	b_1
a_4	b_4

$=$

A	B
a_1	b_1
a_2	b_1
a_2	b_2
a_3	b_1
a_4	b_4

DIFFERENCE

Notation: $R_1 - R_2$

- outputs all tuples in R_1 and not in R_2
- both relations must have the same schema!
- output schema: same as input

A	B
a_1	b_1
a_2	b_1
a_2	b_2

$-$

A	B
a_1	b_1
a_3	b_1
a_4	b_4

$=$

A	B
a_2	b_1
a_2	b_2

CROSS-PRODUCT

Notation: $R_1 \times R_2$

- matches each tuples in R_1 with each tuple in R_2
- input schema: $R_1(A_1, A_2, \ldots, A_n)$, $R_2(B_1, B_2, \ldots, B_m)$
- output schema: $R(A_1, \ldots, A_n, B_1, \ldots, B_m)$

Example

- $Person \times Department$ —

```
SELECT *
FROM Person,Department;
```

CROSS-PRODUCT: EXAMPLE

Person

SSN	name
934729837	Paris
123123645	John

Dependent

depSSN	depname
934729837	Helen
934729837	Bob

$$Person \times Dependent$$

SSN	name	depSSN	depname
934729837	Paris	934729837	Helen
123123645	John	934729837	Bob
934729837	Paris	934729837	Bob
123123645	John	934729837	Helen

RENAMING

Notation: $\rho_{A_1,A_2,\ldots,A_n}(R)$

- does not change the instance, only the schema!
- input schema: $R(B_1, B_2, \ldots, B_n)$
- output schema: $R(A_1, \ldots, A_n)$

Why is it necessary?

named perspective: when joining relations, we need to distinguish between attributes with the same name!

RENAMING: EXAMPLE

Person

SSN	name
934729837	Paris
123123645	John

Dependent

SSN	name
934729837	Helen
934729837	Bob

$$Person \times \rho_{depSSN,depname}\ (Dependent)$$

SSN	name	depSSN	depname
934729837	Paris	934729837	Helen
123123645	John	934729837	Bob
934729837	Paris	934729837	Bob
123123645	John	934729837	Helen

Derived Operations

INTERSECTION

Notation: $R_1 \cap R_2$

- outputs all tuples in R_1 and R_2
- output schema: same as input
- can be expressed as: $R_1 - (R_1 - R_2)$

```
SELECT R.A, R.B
FROM R,S
WHERE R.A = S.A
AND R.B = S.B;
```

R

A	B
a_1	b_1
a_2	b_1
a_2	b_2

$\cap$

S

A	B
a_1	b_1
a_3	b_1
a_4	b_4

$=$

A	B
a_1	b_1

JOIN (THETA JOIN)

Notation: $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$

- cross-product followed by a selection
- θ can be any boolean-valued condition
- might have less tuples than the cross-product!

```
SELECT *
FROM R1, R2
WHERE θ;
```

THETA JOIN: EXAMPLE

Person

SSN	name	age
934729837	Paris	26
123123645	John	22

Dependent

dSSN	dname	dage
934729837	Helen	23
934729837	Bob	28

$$Person \bowtie_{Person.age > Dependent.dage} Dependent$$

SSN	name	age	dSSN	dname	dage
934729837	Paris	26	934729837	Helen	23

EQUI-JOIN

Notation: $R_1 \bowtie_\theta R_2$

- special case of join where the condition θ contains only equalities between attributes
- output schema: same as the cross-product

Example for $R(A, B), S(C, D)$

- $R \bowtie_{B=C} S$
- output schema: $T(A, B, C, D)$

```
SELECT *
FROM R, S
WHERE R.B = S.C;
```

NATURAL JOIN

Notation: $R_1 \bowtie R_2$

- equi-join on all the **common** fields
- the output schema has one copy of each common attribute

Person

SSN	name	age
934729837	Paris	26
123123645	John	22

Dependent

SSN	dname
934729837	Helen
934729837	Bob

```
SELECT SSN,name,age,dname
FROM Person P,
Department D
WHERE P.SSN = D.SSN ;
```

$Person \bowtie Dependent$

SSN	name	age	dname
934729837	Paris	26	Helen
934729837	Paris	26	Bob

NATURAL JOIN

Natural Join $R \bowtie S$

- Input schema: $R(A, B, C, D), S(A, C, E)$
  - Output schema: *T(A, B, C, D, E)*
- Input schema: $R(A, B, C), S(D, E)$
  - Output schema: *T(A, B, C, D, E)*
- Input schema: $R(A, B, C), S(A, B, C)$
  - Output schema? *T(A, B, C,)*

SEMI-JOIN

Notation: $R_1 \ltimes R_2$

- natural join followed by projection on the attributes of R_1

Example:

- $R(A, B, C), S(B, D)$
- $R \ltimes S = \pi_{A,B,C}(R \bowtie S)$
- output schema: *T(A, B, C)*

```
SELECT A,B,C
FROM R, S
WHERE R.B = S.B ;
```

DIVISION

Notation: R_1 / R_2

- suppose $R_1(A, B)$ and $R_2(B)$
- the output contains all values **a** such that for every tuple (**b**) in R_2, tuple **(a, b)** is in R_1
- output schema: *R(A)*

DIVISION: EXAMPLE

A

A	B
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1

$\mathbf{B_1}$

B
b_2
b_3
b_1

$\mathbf{B_2}$

B
b_1

$\boldsymbol{A / B_1}$

A
a_1

$\boldsymbol{A / B_2}$

A
a_1
a_2

EXTENDING RELATIONAL ALGEBRA

GROUP BY AGGREGATE

- is part of the so-called extended RA
- helps us to compute counts, sums, min, max, …

Examples

- What is the average age of the customers?
- How many people bought an iPad?

GROUP BY AGGREGATE

Notation: $\gamma_{X,Agg(Y)}(R)$

- group by the attributes in X
- aggregate the attribute in Y
  - SUM, COUNT, AVG (average), MIN, MAX

- Output schema: X + an extra (numerical) attribute

EXAMPLE

Person

SSN	name	age
934729837	Paris	24
123123645	John	30
384475687	Arun	21

$$\gamma_{AVG(age)}(Person)$$

AVG(age)
25

```sql
SELECT AVG(age)
FROM Person ;
```

EXAMPLE

Person

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	21	206-473-8221

```sql
SELECT SSN,
  COUNT(phoneNumber)
FROM Person
GROUP BY SSN;
```

$$\gamma_{SSN,COUNT(phoneNumber)}(Person)$$

SSN	COUNT(phoneNumber)
934729837	2
123123645	1
384475687	1

Constructing RA Queries

COMBINING RA OPERATORS

- We can build more complex queries by combining RA operators together

  e.g. standard algebra: $(x + 1) * y - z^2$

- There are 3 different notations:
  - sequence of assignment statements
  - expressions with operators
  - expression trees

COMBINING RA OPERATORS

Input schema: $R(B, C), S(A, B)$

- expressions with operators

  $$\pi_A(\sigma_{C=1}(R) \bowtie S)$$

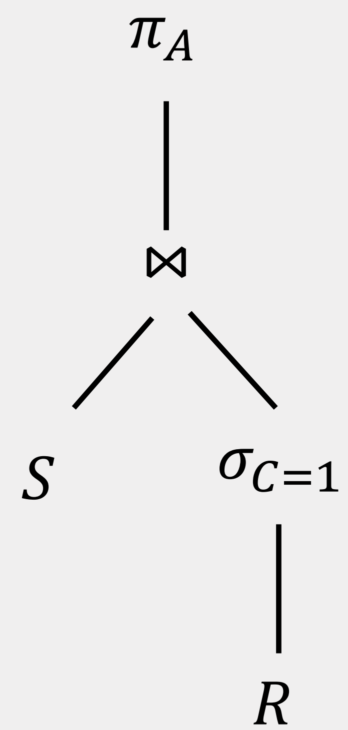
- sequence of assignment statements

  $$R' = \sigma_{C=1}(R)$$

  $$R'' = R' \bowtie S$$

  $$R''' = \pi_A(R'')$$

- expression trees



EXPRESSIVE POWER OF RA

- RA cannot express transitive closure!

Edges

From	**To**
a	b
b	c
a	d
c	d

Transitive closure computes all pairs of nodes connected by a directed path

Relational Algebra: Examples

CS 564- Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

EXAMPLE DB: SAILORS

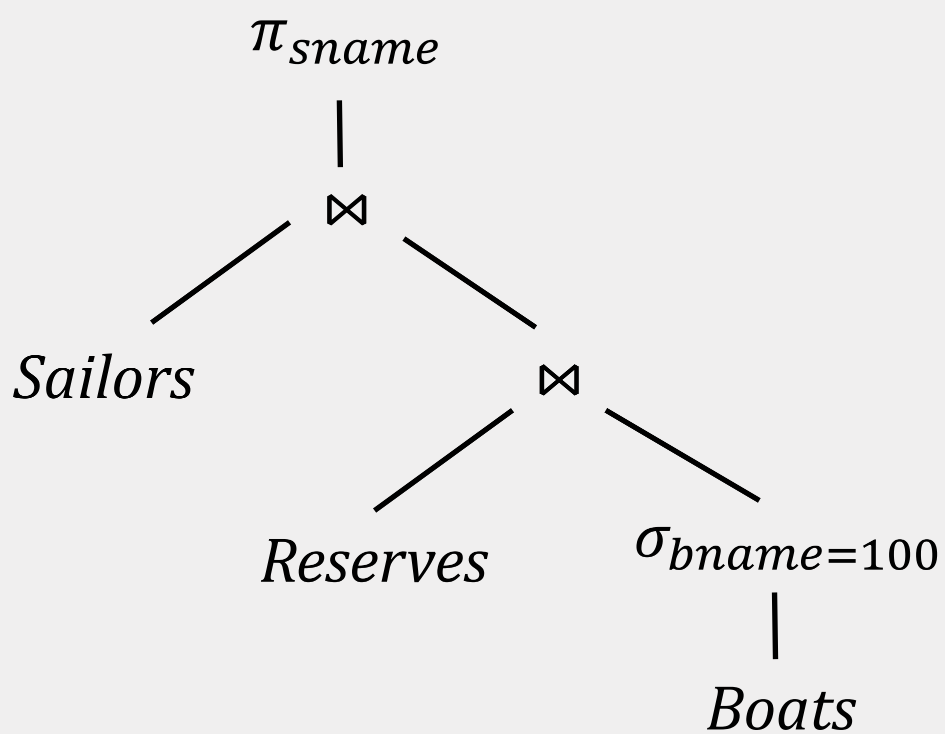
Sailors (<u>sid</u>, sname, rating, age)

Reserves (<u>sid</u>, <u>bid</u>, <u>day</u>)

Boats (<u>bid</u>, bname, color)

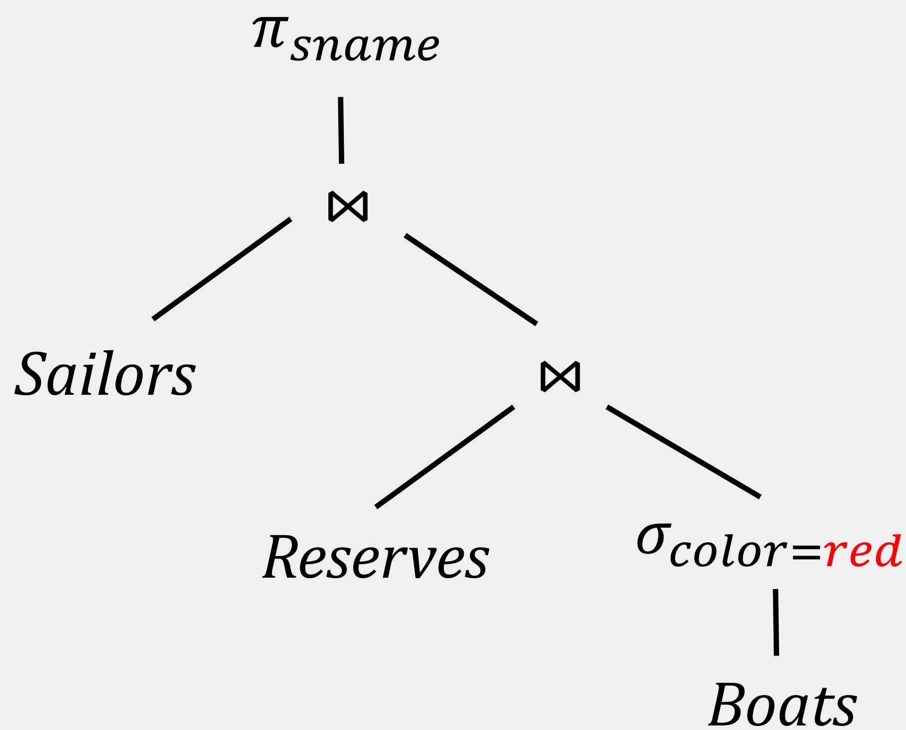
Example DB: Sailors

Q1: *What are the names of the sailors who have reserved boat with name "100" ?*



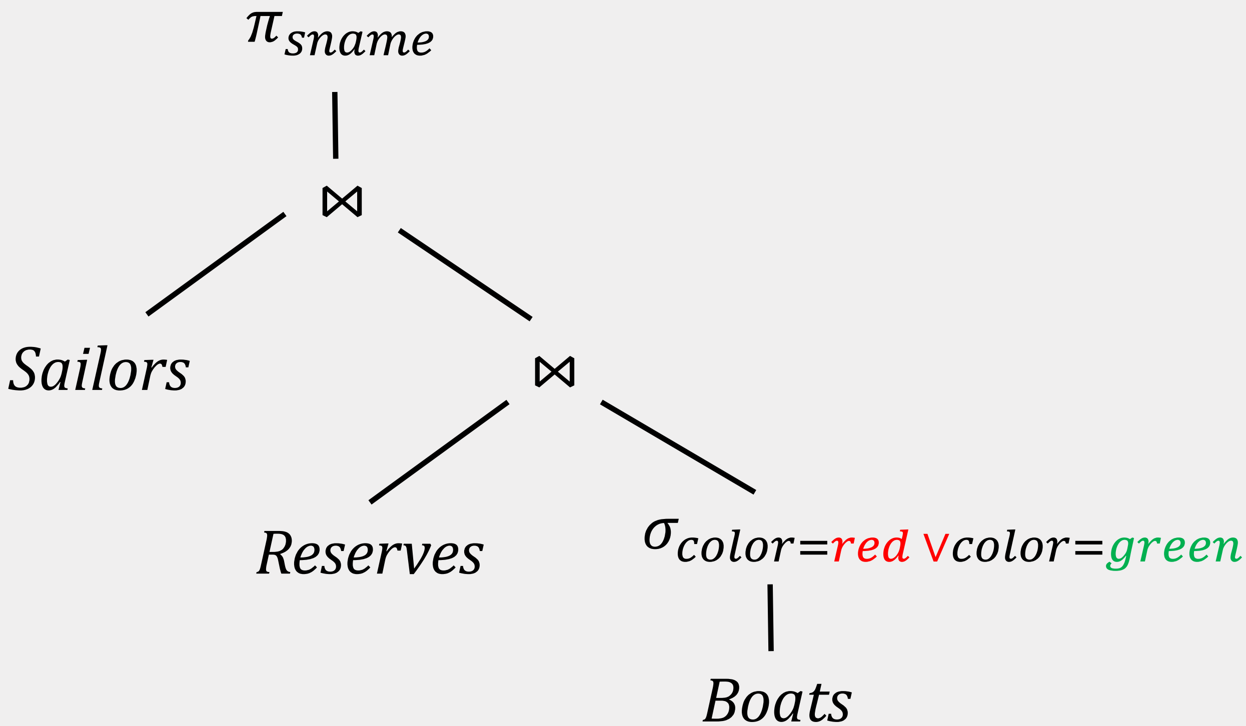
EXAMPLE DB: SAILORS

Q2: *What are the names of the sailors who have reserved a red boat ?*



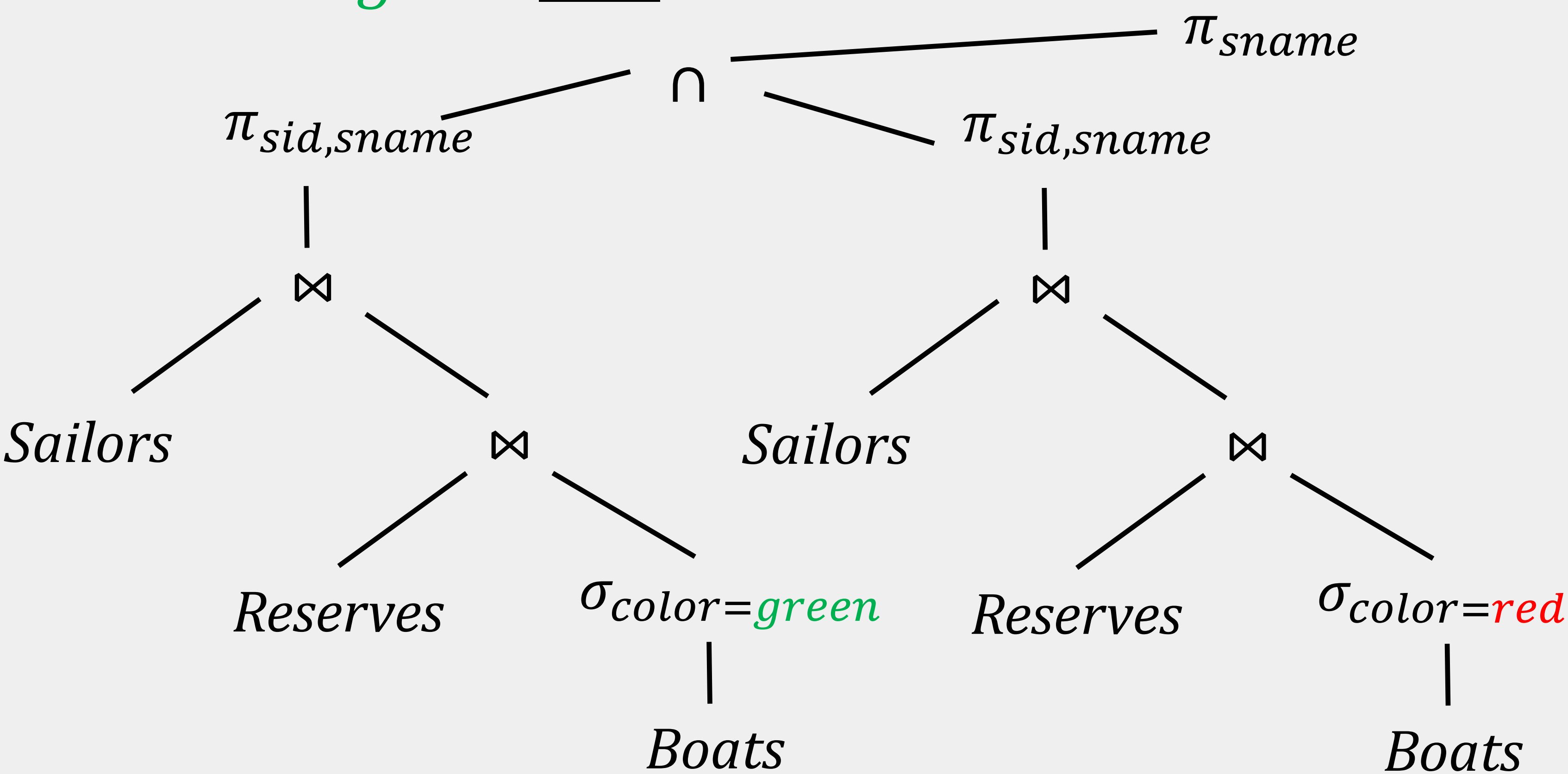
Example DB: Sailors

Q3: *What are the names of the sailors who have reserved a green **or** red boat ?*



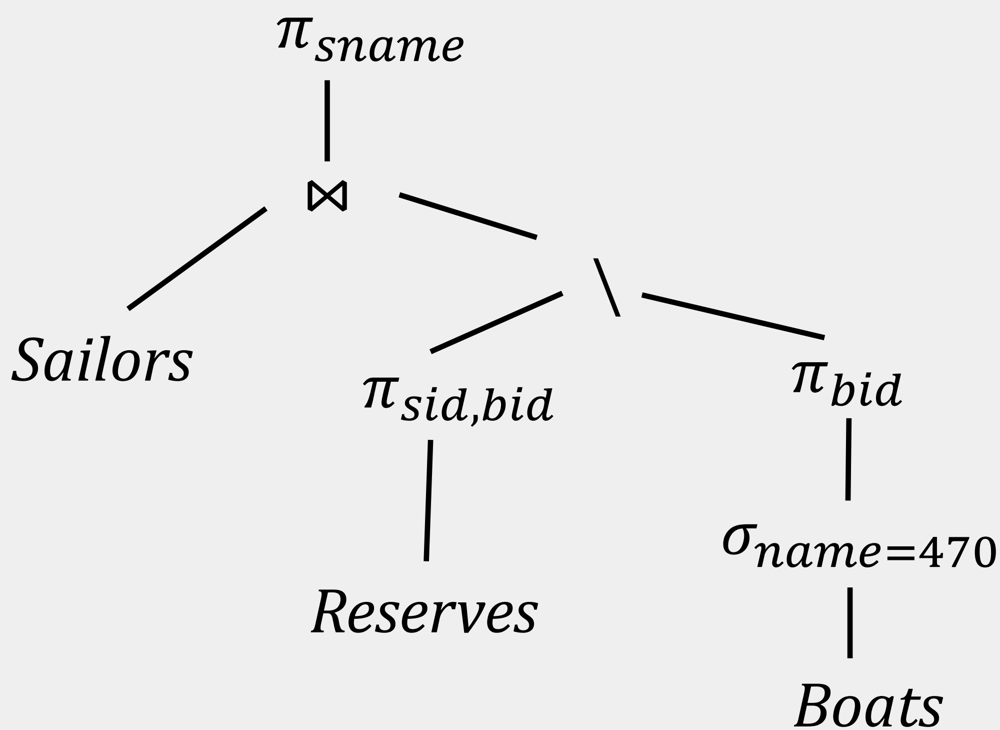
Example DB: Sailors

Q4: *What are the names of the sailors who have reserved a* *green* ***and*** *red* *boat ?*



EXAMPLE DB: SAILORS

Q5: *Find the names of the sailors who have reserved all boats with name "470".*



Example DB: Products

Product (<u>pid</u>,name,price,category,maker-cid)

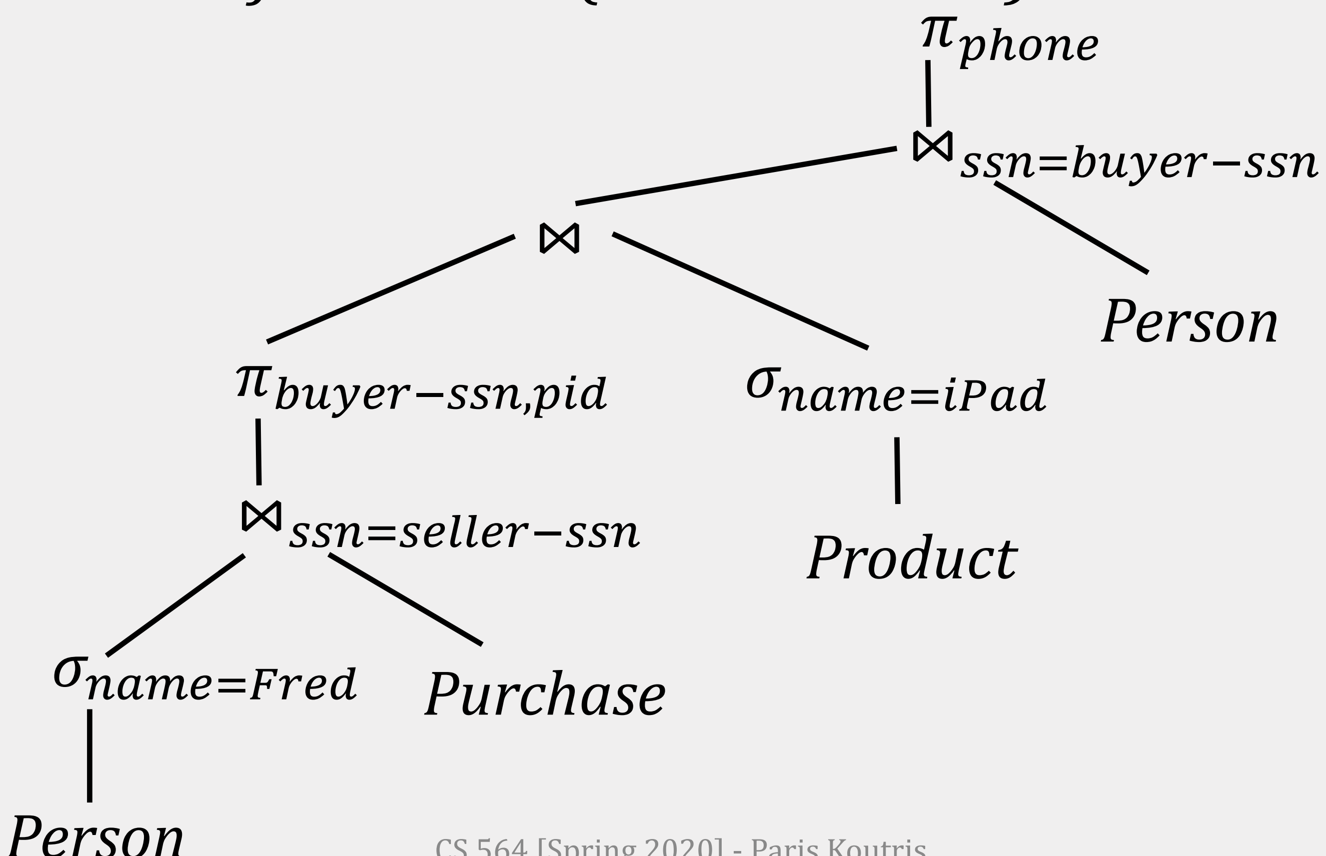
Purchase (buyer-ssn, seller-ssn, store, pid)

Company (<u>cid</u>, name, country)

Person (<u>ssn</u>,name, phone, city)

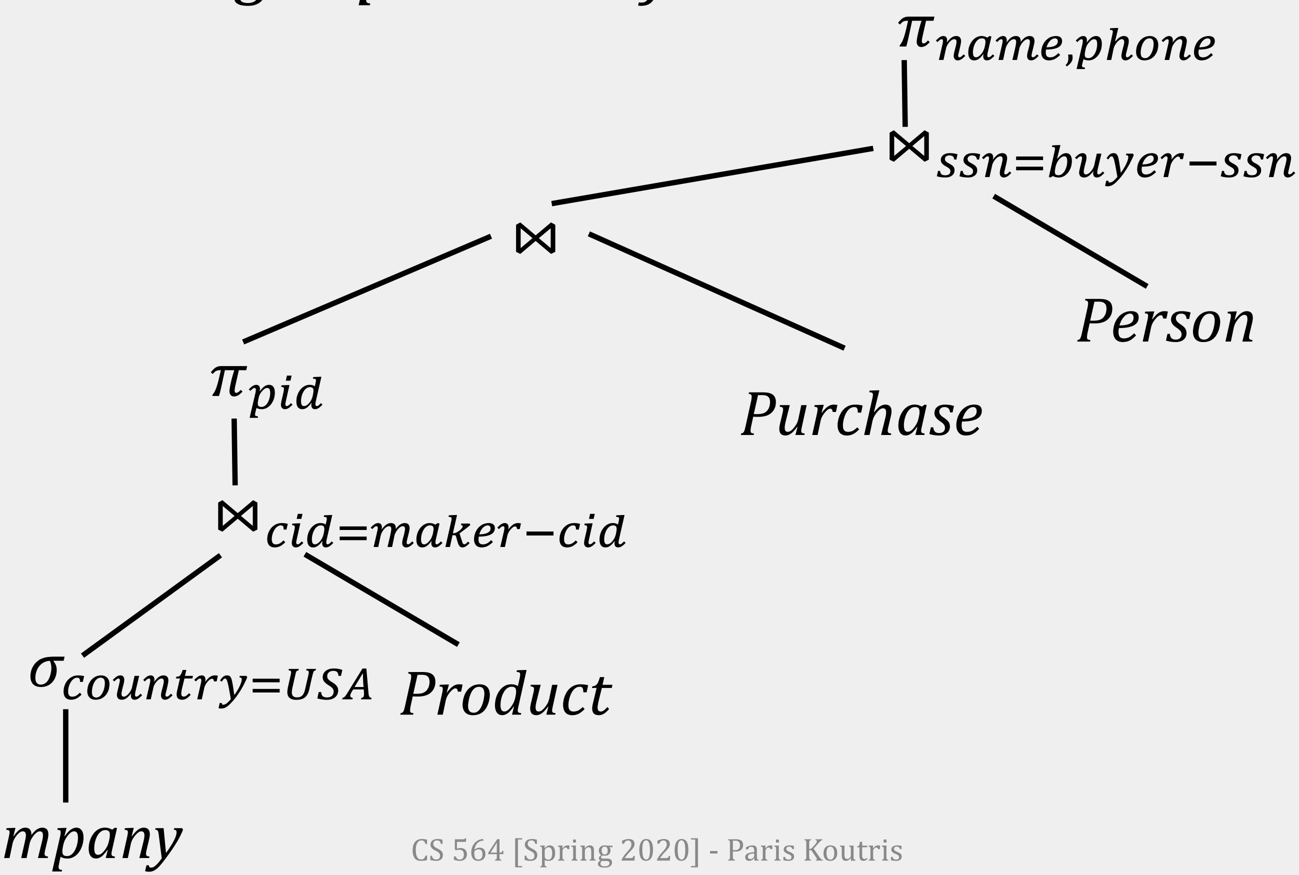
Example DB: Products

Q6: *Find the phone numbers of the people who have bought iPads from Fred (the salesman).*



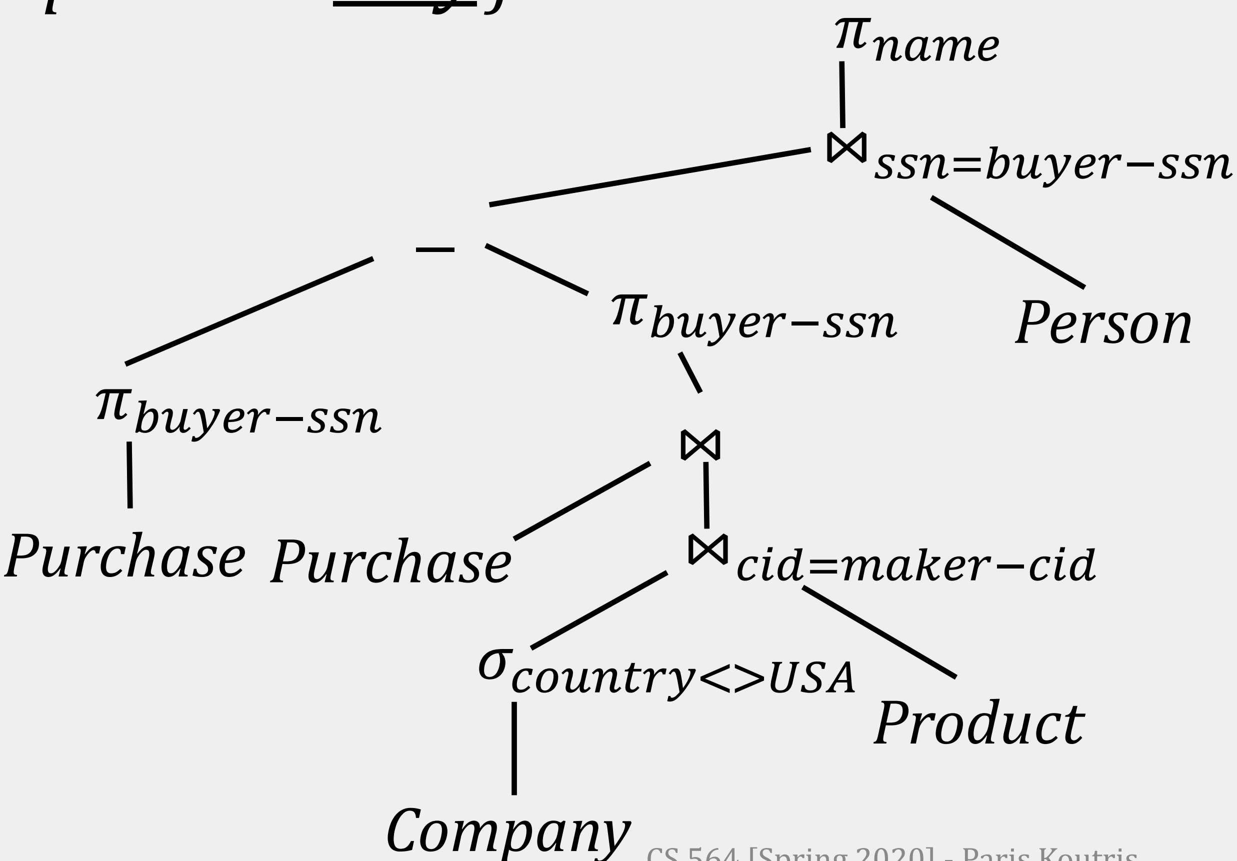
Example DB: Products

Q7: *Find the names and phone numbers of the people who have bought products from the USA.*



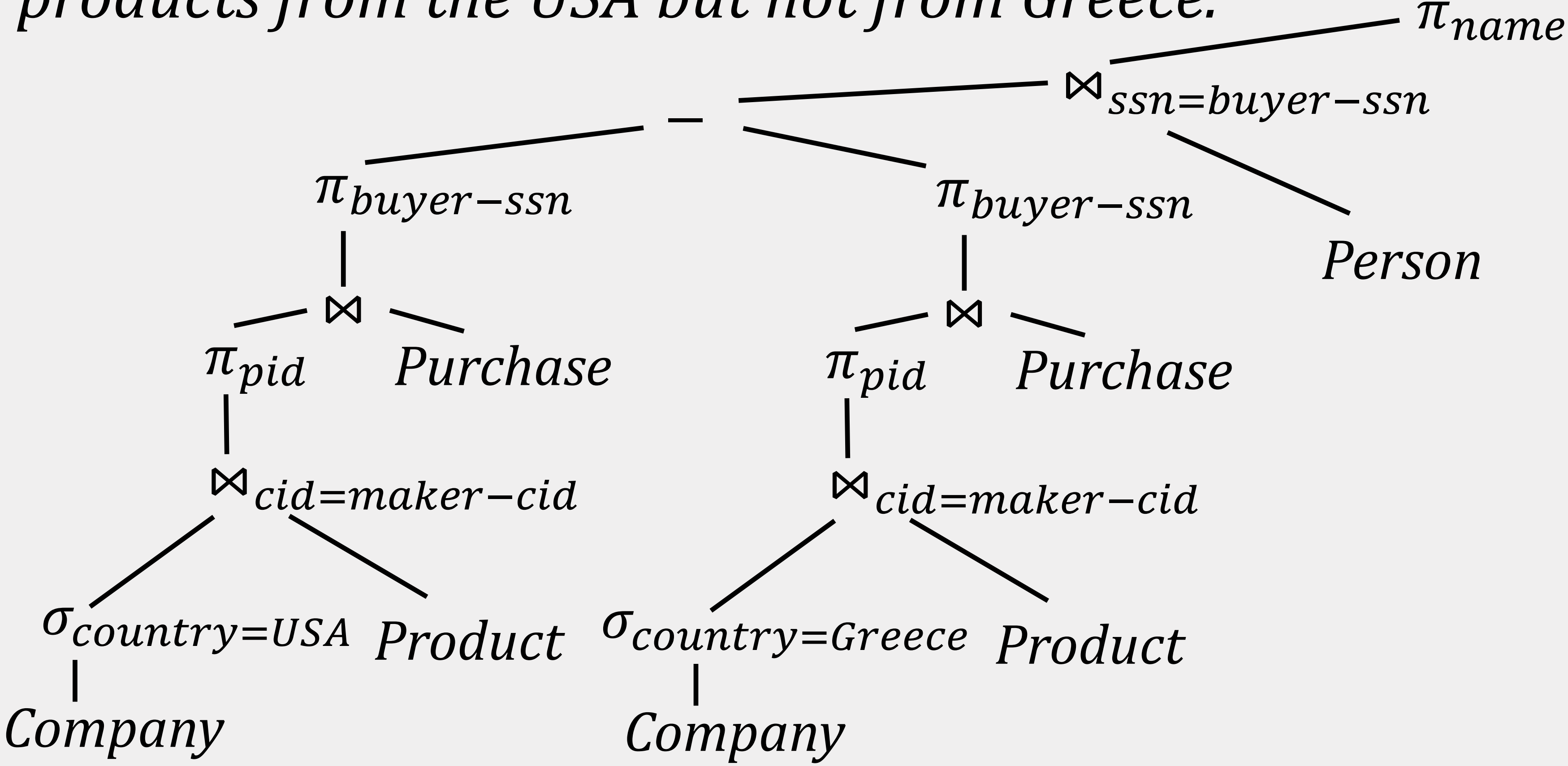
EXAMPLE DB: PRODUCTS

Q8: *Find the names of the people who have bought products* ***<u>only</u>*** *from the USA.*



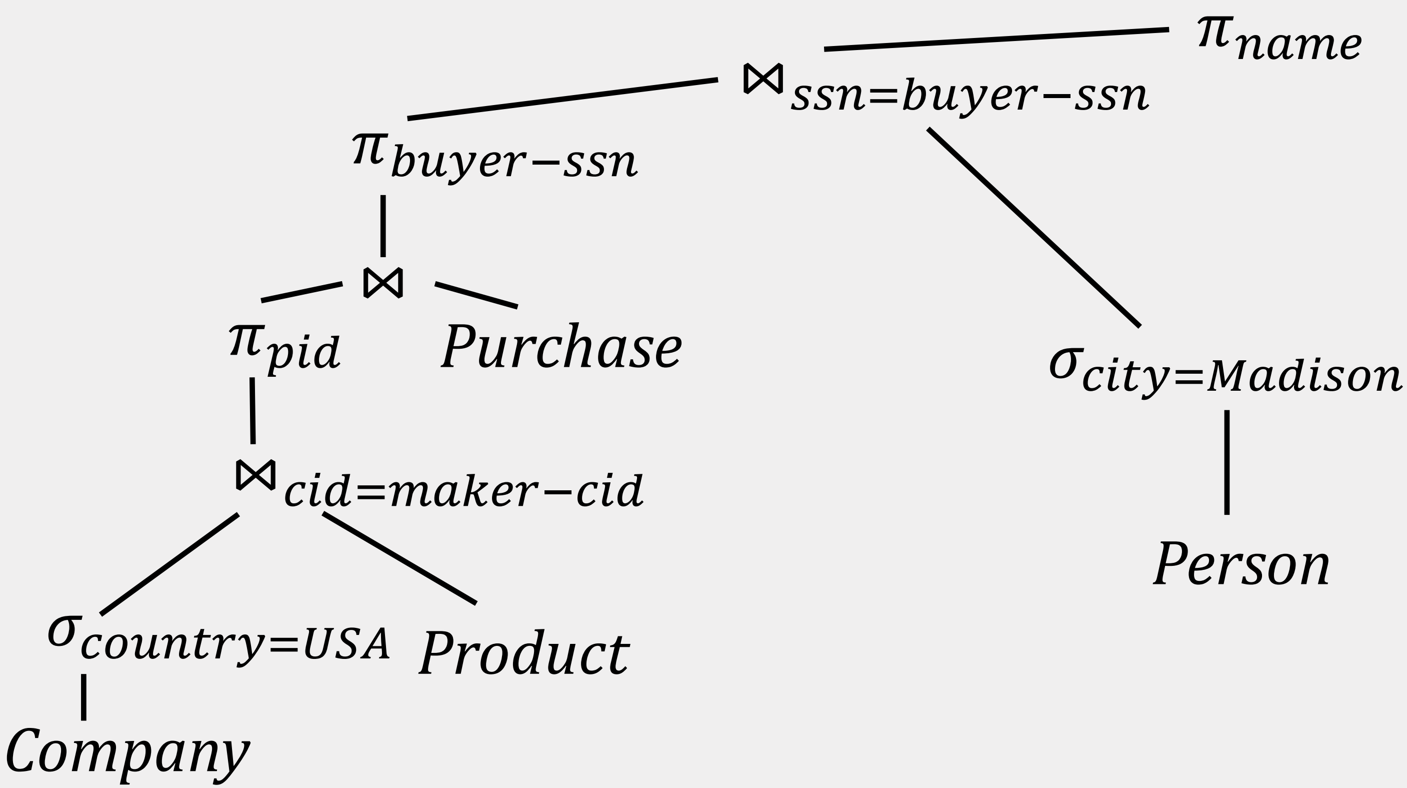
EXAMPLE DB: PRODUCTS

Q9: *Find the names of the people who have bought products from the USA but not from Greece.*



EXAMPLE DB: PRODUCTS

Q10: *Find the names of the people who have bought products from the USA and live in Madison.*



EXAMPLE DB: FLIGHTS

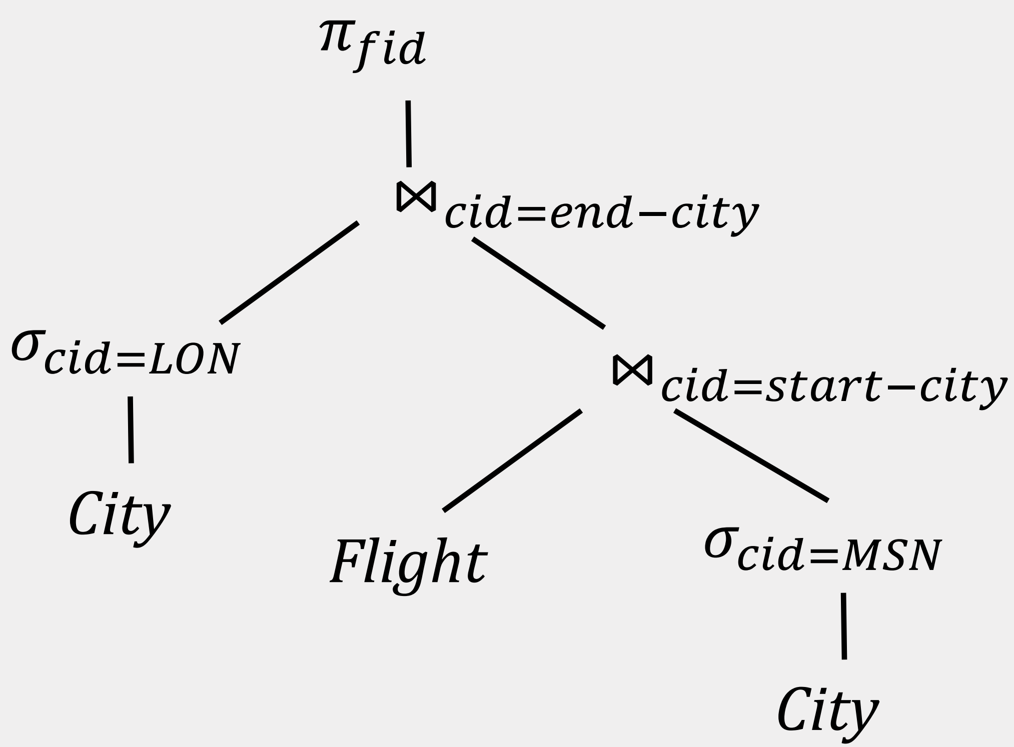
City (<u>cid</u>, name, population)

Flight (<u>fid</u>, length,start-city,end-city,aid)

Airline (<u>aid</u>, name, profit)

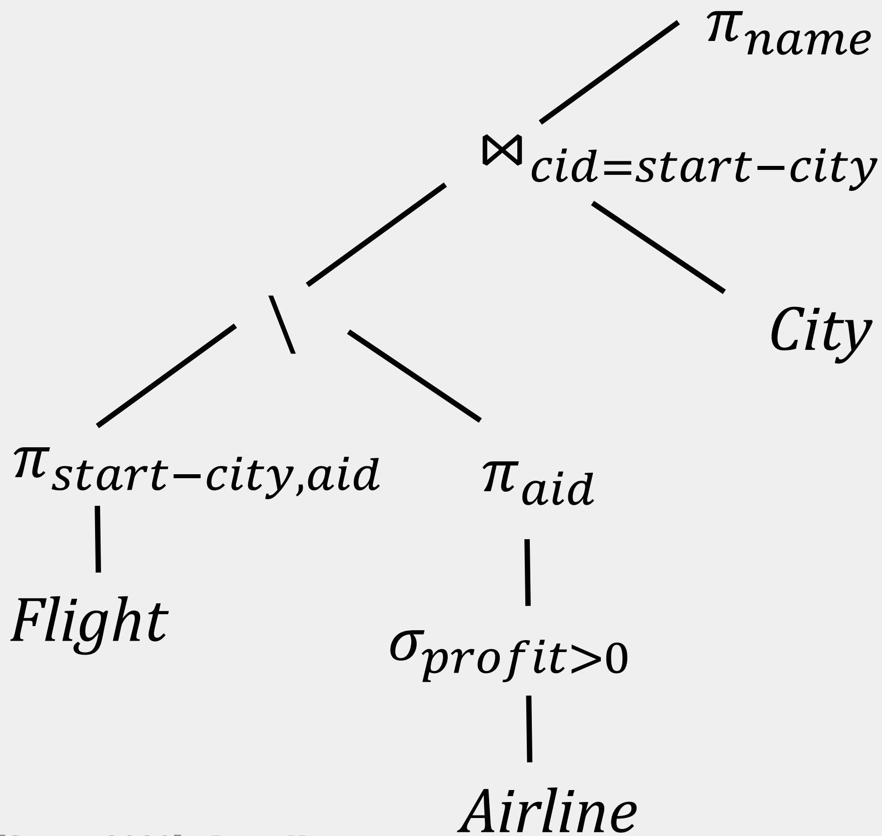
Example DB: Flights

Q11: *Find the flight ids for flights that start in a city with id "MSN" and end in a city with id "LON".*



EXAMPLE DB: FLIGHTS

Q12: *Find the names of the cities that have a flight for* ***<u>every</u>*** *airline with profit more than 0.*



STORING DATA: DISK AND FILES

CS 564- Fall 2018

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

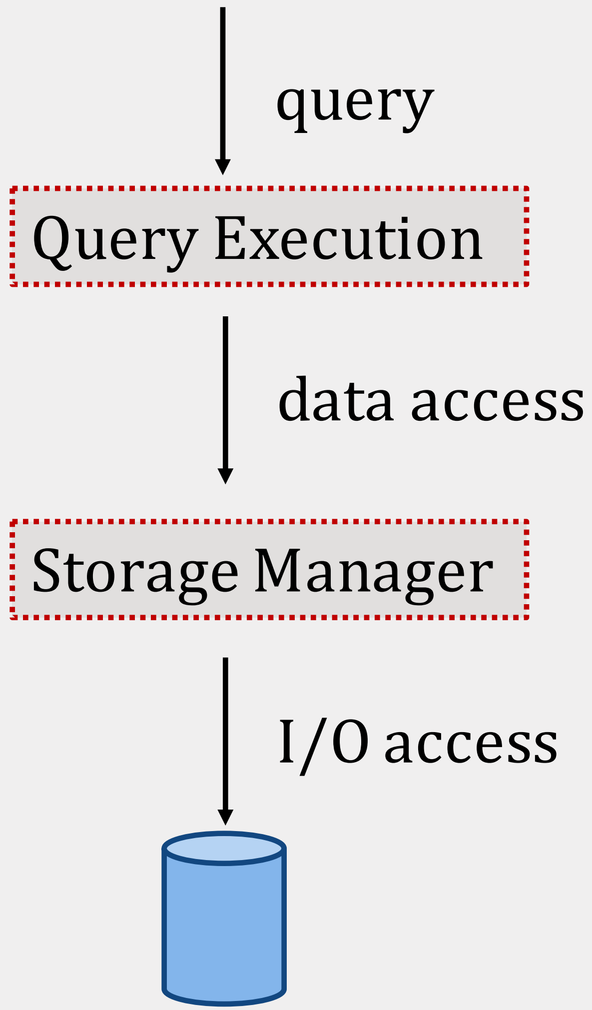
How does a DBMS store data?

- disk, SSD, main memory

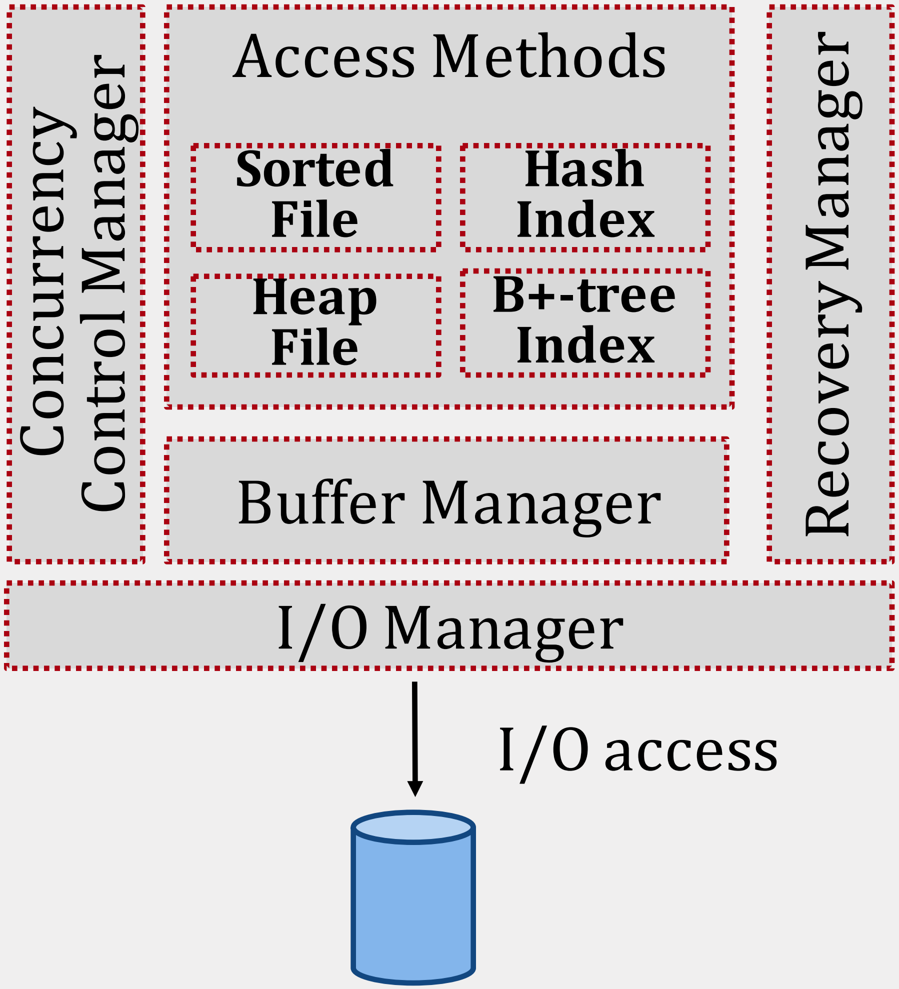
The **buffer manager**

- controls how the data moves between main memory and disk
- uses various replacement policies (LRU, Clock)

ARCHITECTURE OF A DBMS



ARCHITECTURE OF STORAGE MANAGER



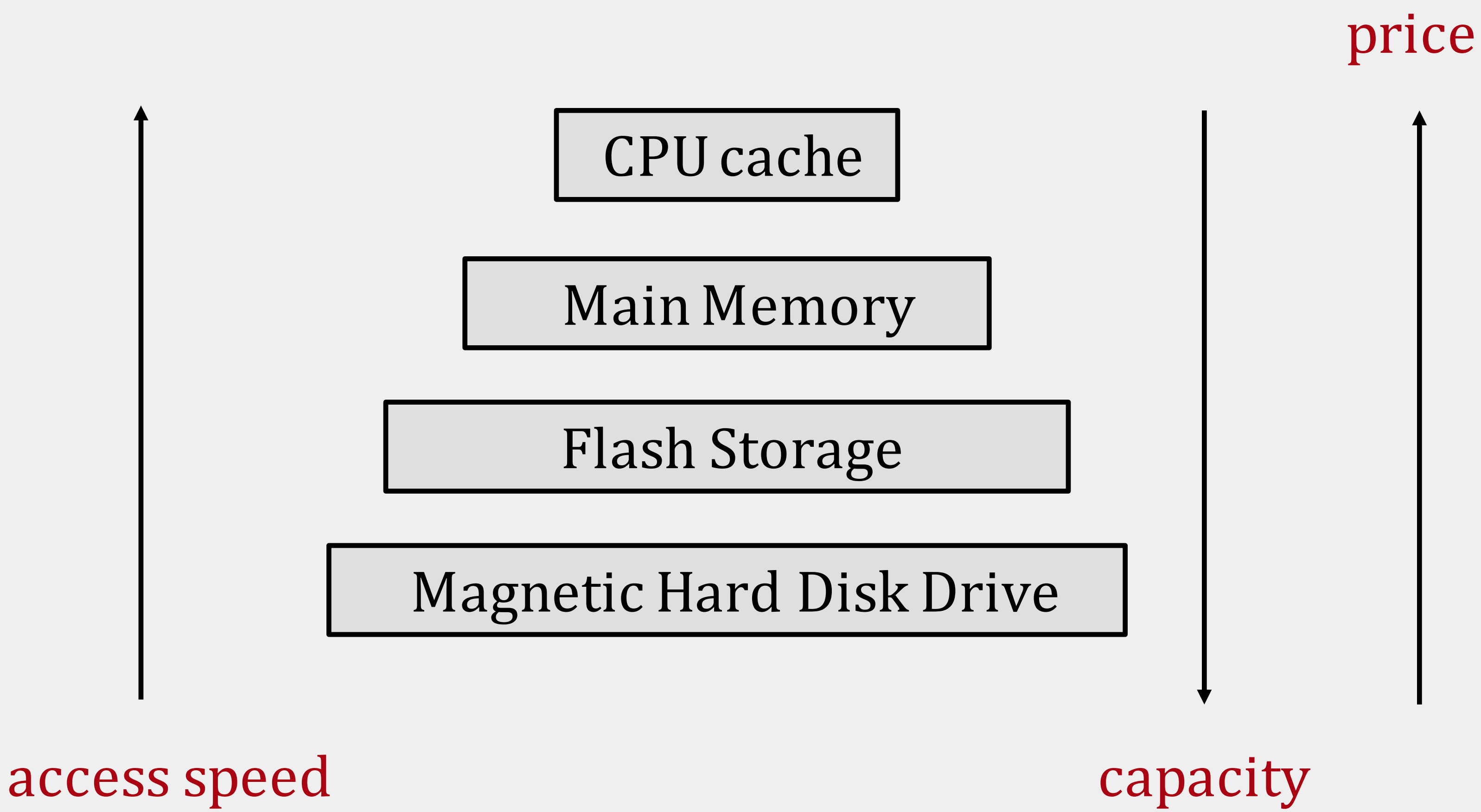
DATA STORAGE

- How does a DBMS store and access data?
  – main memory (fast, temporary)
  – disk (slow, permanent)

- How do we move data from disk to main memory?
  – buffer manager

- How do we organize relational data into files?
  – next lecture!

MEMORY HIERARCHY



WHY NOT MAIN MEMORY?

- Relatively high cost
- Main memory is not persistent!
- Typical storage hierarchy:
  - **Primary storage**: main memory (RAM) for currently used data
  - **Secondary storage**: disk for the main database
  - **Tertiary storage**: tapes for archiving older versions of the data

DISK

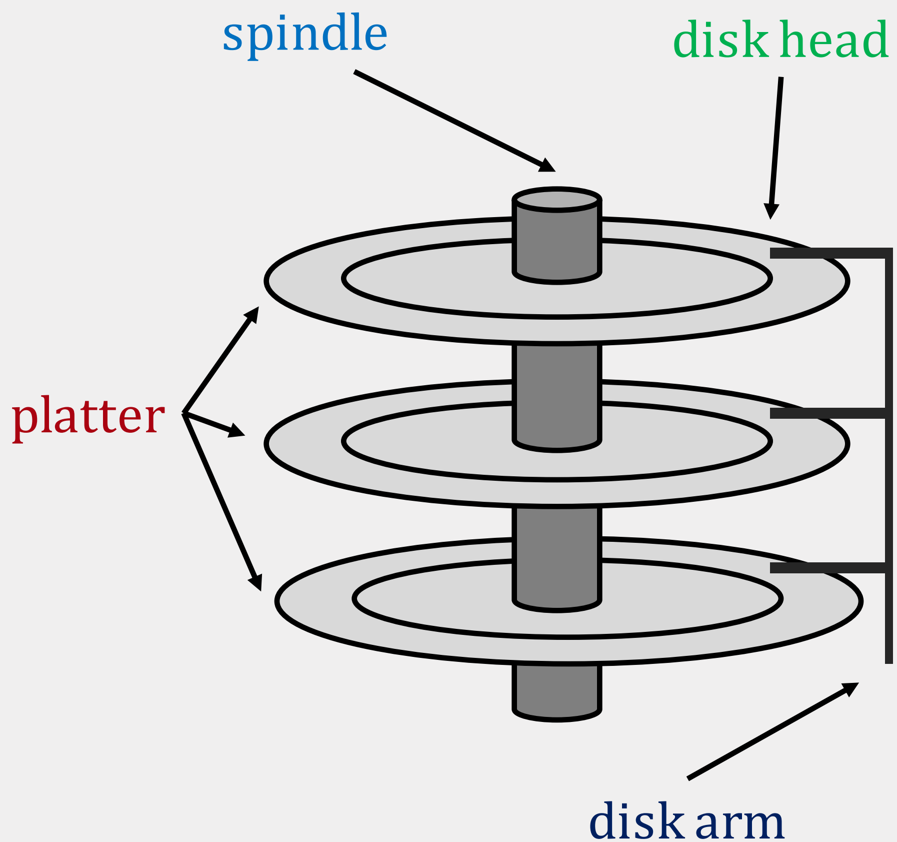
DISKS

- Secondary storage device of choice
- Data is stored and retrieved in units called <u>disk blocks</u>
- The time to retrieve a disk block varies depending upon location on disk (unlike RAM)

The placement of blocks on disk has major impact on DBMS performance!

COMPONENTS OF DISKS

- platter: circular hard surface on which data is stored by inducing magnetic changes
- spindle: axis responsible for rotating the platters
- disk head: mechanism to read or write data
- disk arm: moves to position a head on a desired track of the platter

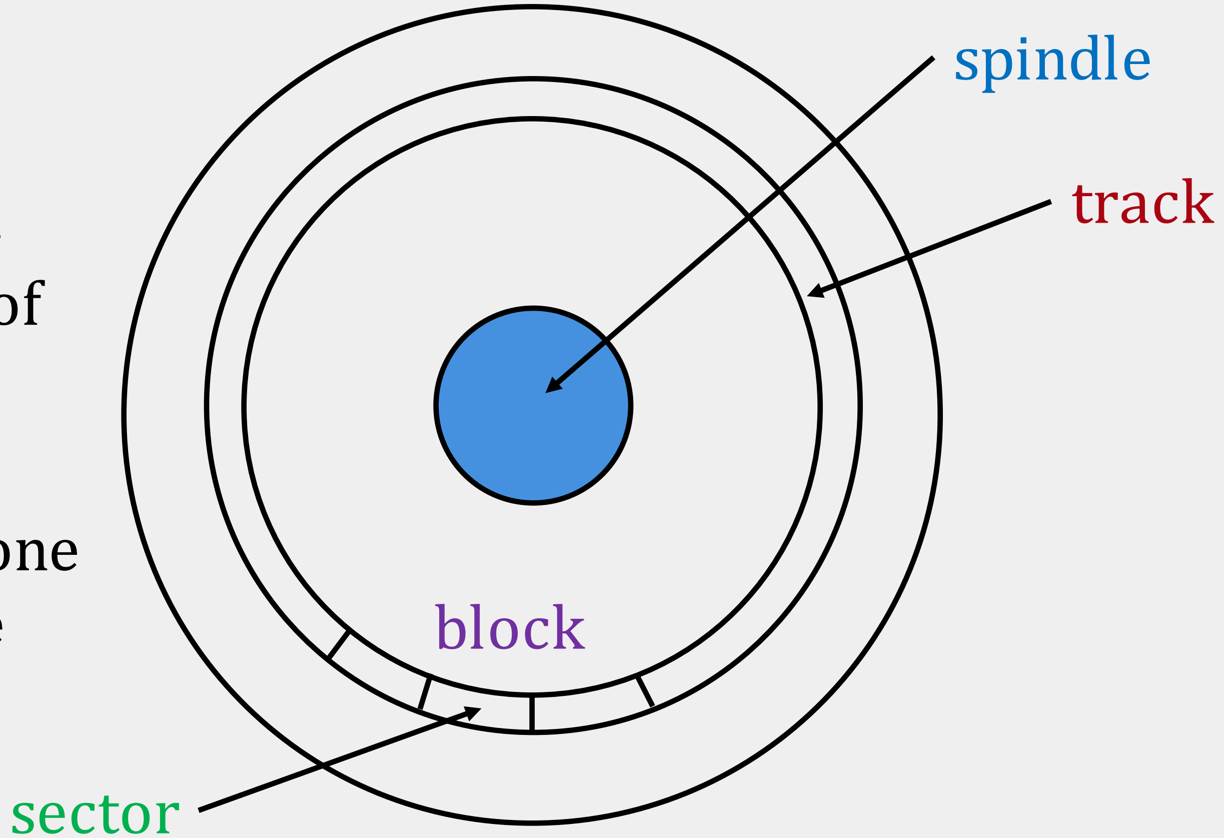


RPM (**R**otations **P**er **M**inute)

- 7200 RPM – 15000 RPM

COMPONENTS OF DISKS

- data is encoded in concentric circles of sectors called tracks
- block size: multiple of sector size (which is fixed)
- at any time, exactly one head can read/write



ACCESSING THE DISK

- unit of read or write: block size
- once in memory, we refer to it as a **page**
- typically: 4k or 8k or 16k

access time = **rotational delay** + seek time + transfer time

ACCESSING THE DISK (1)

access time = rotational delay + seek time + transfer time

rotational delay: time to wait for sector to rotate under the disk head

- typical delay: *0–10 ms*
- maximum delay = 1 full rotation
- average delay ~ half rotation

RPM	**Average delay**
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00

ACCESSING THE DISK (2)

access time = rotational delay + **seek time** + transfer time

seek time: time to move the arm to position disk head on the right track

- typical seek time: ~ *9 ms*
- ~ *4 ms* for high-end disks

ACCESSING THE DISK (3)

access time = rotational delay + seek time + **transfer time**

data transfer time: time to move the data to/from the disk surface

- typical rates: *~100 MB/s*
- the access time is dominated by the seek time and rotational delay!

EXAMPLE: SPECS

	Seagate HDD
Capacity	*3 TB*
RPM	*7,200*
Average Seek Time	*9 ms*
Max Transfer Rate	*210 MB/s*
# Platters	*3*

What are the I/O rates for block size *4 KB* and:

- random workload (~ *0.3 MB/s*)
- sequential workload (~ *210 MB/s*)

EXAMPLE: RANDOM WORKLOAD

	Seagate HDD
Capacity	*3 TB*
RPM	*7,200*
Average Seek Time	*9 ms*
Max Transfer Rate	*210 MB/s*
# Platters	*3*

For a 4KB block:

- rotational delay = 4.17 ms
- seek time = 9 ms
- transfer time = (4KB) / (210 MB/s) ~ 0.019 ms
- total time per block = 13.1 ms
- I/O rate = (4KB) / (13.1 ms) ~ 0.3 MB/s

ACCESSING THE DISK

- Blocks in a file should be arranged sequentially on disk to minimize seek and rotational delay!

- next block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder

MANAGING DISK SPACE

- The disk space is organized into files
- Files are made up of pages
- Pages contain records

- Data is allocated/deallocated in increments of pages
- Logically close pages should be nearby in the disk

SSD (Solid State Drive)

- SSDs use flash memory
- No moving parts (no rotate/seek motors)
  - eliminates seek time and rotational delay
  - very low power and lightweight
- Data transfer rates: 300-600 MB/s
- SSDs can read data (sequential **or** random) very fast!

SSDs

- Small storage (0.1-0.5x of HDD)
- expensive (20x of HDD)
- Writes are much more expensive than reads (10x)
- Limited lifetime
  - 1-10K writes per page
  - the average failure rate is 6 years

Buffer Management

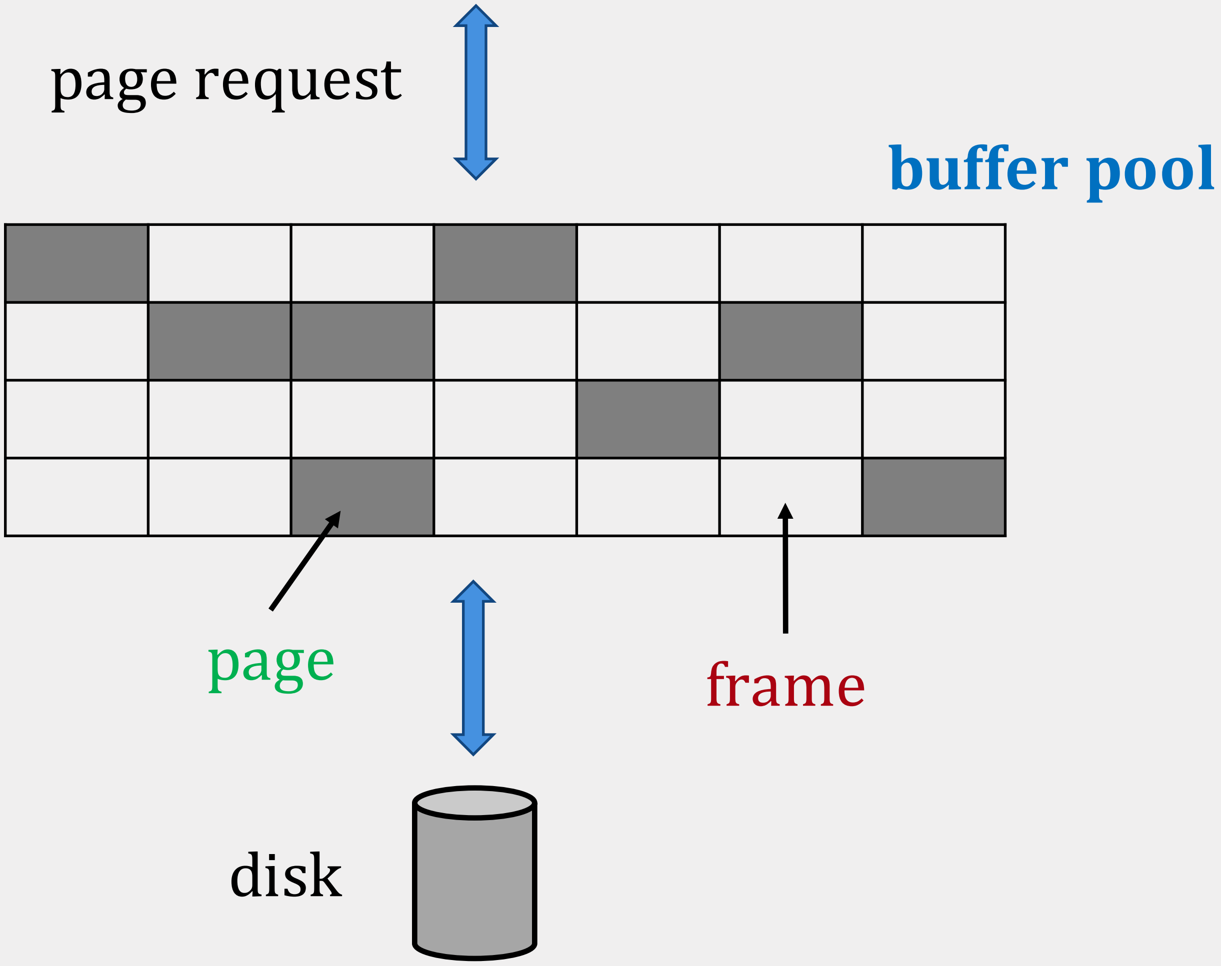
BUFFER MANAGER

- Data must be in RAM for DBMS to operate on it
- All the pages may not fit into main memory

Buffer manager: responsible for bringing pages from disk to main memory as needed

- pages brought into main memory are in the buffer pool
- the buffer pool is partitioned into frames: slots for holding disk pages

BUFFER MANAGER



BUFFER MANAGER: REQUESTS

- **Read (page):** read a page from disk and add to the buffer pool *(if not already in buffer)*

- **Flush (page):** evict page from buffer pool & write to disk

- **Release (page):** evict page from buffer pool *without* writing to disk

BOOKKEEPING

Bookkeeping per frame:

- pin count: # current users of the page
  - *pinning* : increment the pin count
  - *unpinning* : decrement the pin count
- dirty bit: indicates if the page has been modified
  - bit = 1 means that the changes to the page must be propagated to the disk

PAGE REQUEST

- Page is in the buffer pool:
  - return the address to the frame
  - increment the pin count
- Page is not in the buffer pool:
  - choose a frame for replacement (with pin count = 0)
  - if frame is dirty, write the page to disk
  - read requested page into chosen frame
  - pin the page and return the address

BUFFER REPLACEMENT POLICY

- How do we choose a frame for replacement?
  - LRU (**L**east **R**ecently **U**sed)
  - Clock
  - MRU (**M**ost **R**ecently **U**sed)
  - FIFO, random, …

- The replacement policy has big impact on # of I/O's (depends on the access pattern)

LRU

LRU (**L**east **R**ecently **U**sed)

- uses a **queue** of pointers to frames that have pin count = 0
- a page request uses frames only from the *head* of the queue
- when a the pin count of a frame goes to 0, it is added to the *end* of the queue

LRU EXAMPLE

	frame	dirty	pincount
1		0	0
2		0	0
3		0	0

priority queue: 1 **|** 2 **|** 3

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	**A**	0	**1**
2		0	0
3		0	0

priority queue: 2 | 3

one I/O to read the page

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	A	**1**	1
2		0	0
3		0	0

priority queue: 2 | 3

no I/O here!

Sequence of requests:

request A, **modify A**, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	A	1	1
2	**B**	0	**1**
3		0	0

priority queue: 3

one I/O to read the page

Sequence of requests:

request A, modify A, **request B**, request B, release A, request C, release B, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	A	1	1
2	B	0	**2**
3		0	0

priority queue: 3

No I/O here
The pincount increases!

Sequence of requests:

request A, modify A, request B, **request B**, release A, request C, release B, request D, modify D, release B, request A, request E

LRU Example

	frame	dirty	pincount
1	A	1	**0**
2	B	0	2
3		0	0

priority queue: 3 | 1

no I/O yet!

Sequence of requests:

request A, modify A, request B, request B, **release A**, request C, release B, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	A	1	0
2	B	0	2
3	**C**	0	**1**

priority queue: 1

one I/O to read the page

Sequence of requests:

request A, modify A, request B, request B, release A, **request C**, release B, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	A	1	0
2	B	0	**1**
3	C	0	1

priority queue: 1

the pincount decreases

Sequence of requests:

request A, modify A, request B, request B, release A, request C, **release B**, request D, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	**D**	0	**1**
2	B	0	1
3	C	0	1

priority queue:

two I/Os: one to write A to disk and one to read D

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, **request D**, modify D, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	D	**1**	1
2	B	0	1
3	C	0	1

priority queue:

no I/O here

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, **modify D**, release B, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	B	0	**0**
3	C	0	1

priority queue: 2

no I/O

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, **release B**, request A, request E

LRU EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	**A**	0	**1**
3	C	0	1

priority queue:

one I/O to read A

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, **request A**, request E

LRU EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	A	0	1
3	C	0	1

priority queue:

The buffer pool is full, the request must wait!

Sequence of requests:

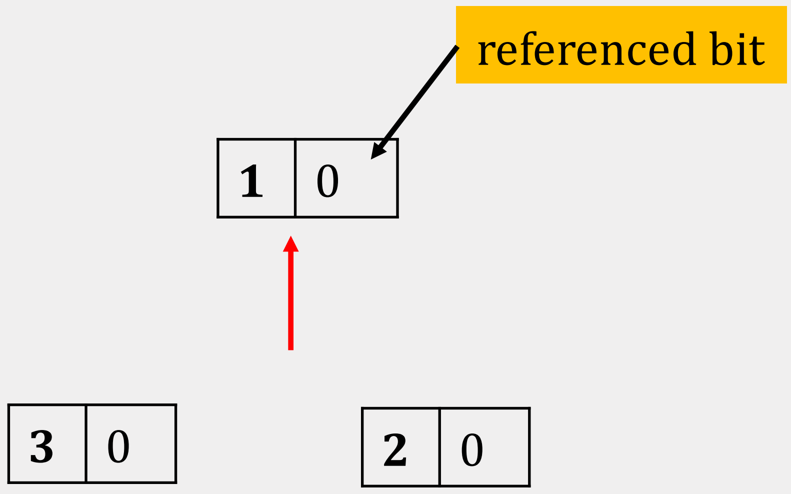
request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, **request E**

CLOCK

- Variant of LRU with lower memory overhead
- The *N* frames are organized into a cycle
- Each frame has a referenced bit that is set to 1 when pin count becomes 0
- A current variable points to a frame
- When a frame is considered:
  - If pin count > 0, increment current
  - If referenced = 1, set to 0 and increment
  - If referenced = 0 and pin count = 0, choose the page

Clock example

	frame	dirty	pincount
1		0	0
2		0	0
3		0	0

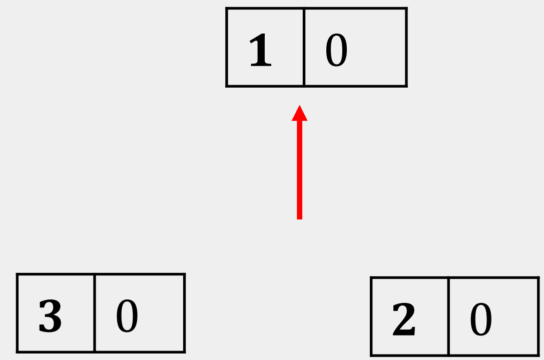


Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	A	0	1
2		0	0
3		0	0

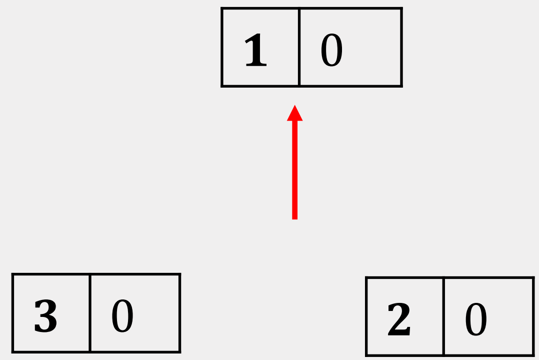


Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	A	1	1
2		0	0
3		0	0

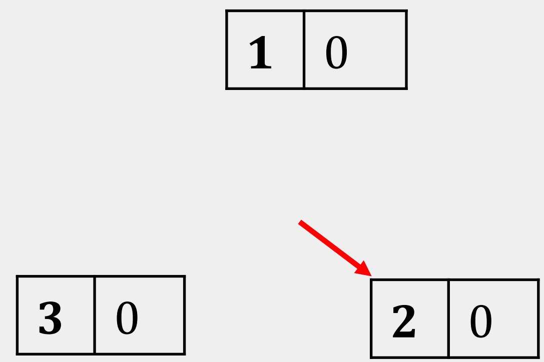


Sequence of requests:

request A, **modify A**, request B, request B, release A, request C, release B, request D, modify D, release B, request A, request E

Clock example

	frame	dirty	pincount
1	A	1	1
2	B	0	1
3		0	0

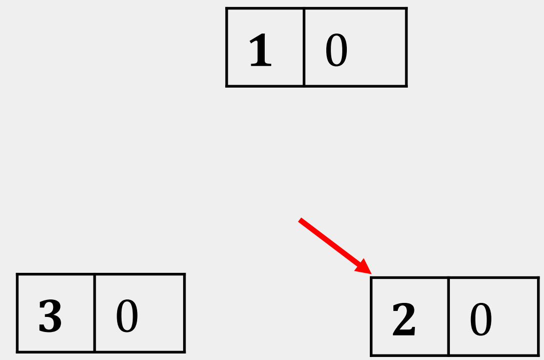


Sequence of requests:

request A, modify A, **request B**, request B, release A, request C, release B, request D, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	A	1	1
2	B	0	2
3		0	0

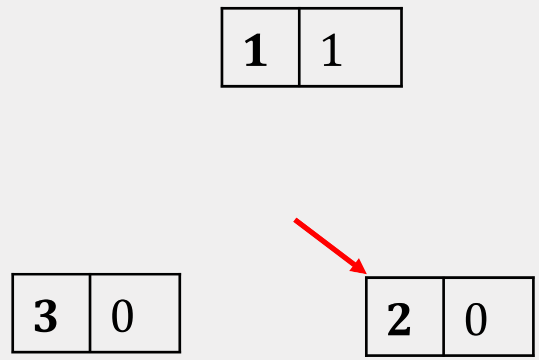


Sequence of requests:

request A, modify A, request B, **request B**, release A, request C, release B, request D, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	A	1	0
2	B	0	2
3		0	0

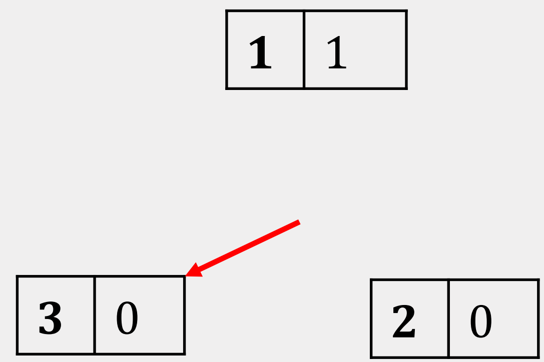


Sequence of requests:

request A, modify A, request B, request B, **release A**, request C, release B, request D, modify D, release B, request A, request E

Clock example

	frame	dirty	pincount
1	A	1	0
2	B	0	2
3	C	0	1

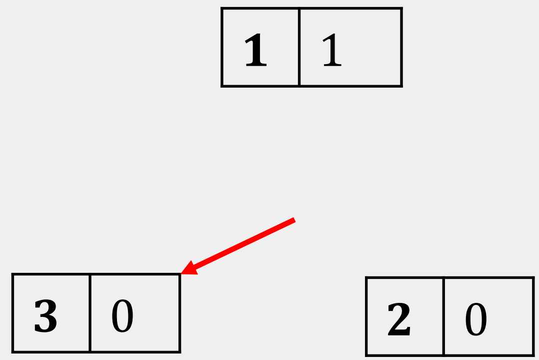


Sequence of requests:

request A, modify A, request B, request B, release A, **request C**, release B, request D, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	A	1	0
2	B	0	1
3	C	0	1

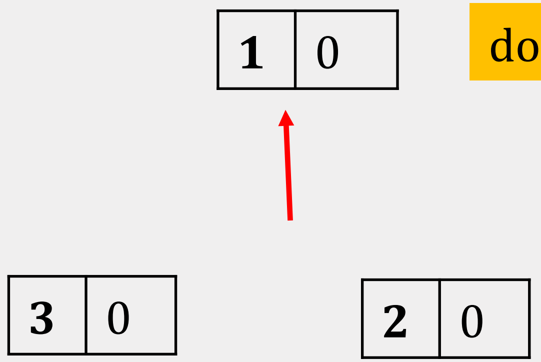


Sequence of requests:

request A, modify A, request B, request B, release A, request C, **release B**, request D, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	D	0	1
2	B	0	1
3	C	0	1



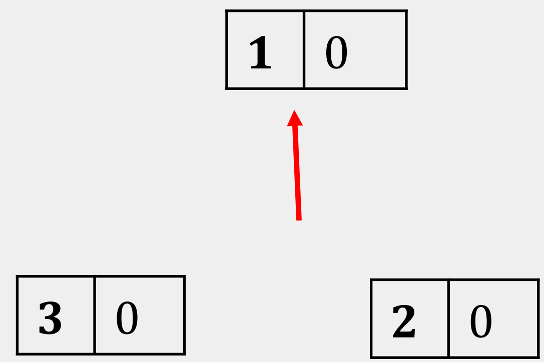
does a full cycle!

Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, **request D**, modify D, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	B	0	1
3	C	0	1

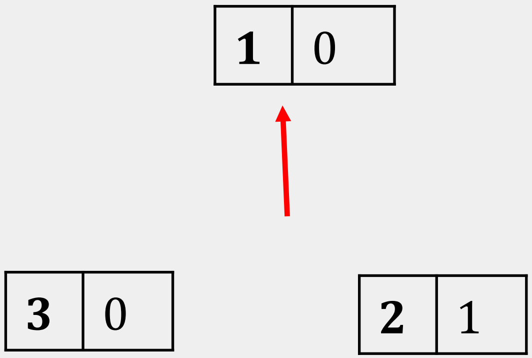


Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, **modify D**, release B, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	B	0	0
3	C	0	1

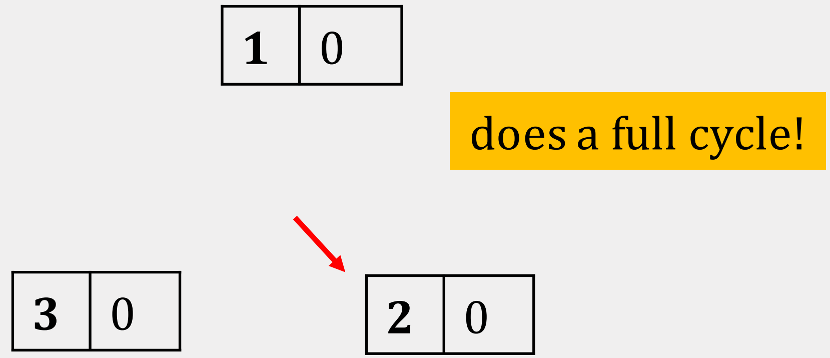


Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, **release B**, request A, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	A	0	1
3	C	0	1

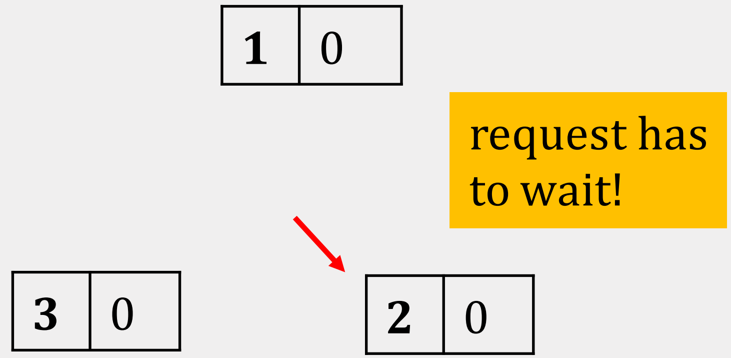


Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, **request A**, request E

CLOCK EXAMPLE

	frame	dirty	pincount
1	D	1	1
2	A	0	1
3	C	0	1



Sequence of requests:

request A, modify A, request B, request B, release A, request C, release B, request D, modify D, release B, request A, **request E**

SEQUENTIAL FLOODING: EXAMPLE

- 3 frames in the buffer pool
- request sequence:
  - A, B, C, D, A, B, C, D, A, B, C, D, …
- With LRU policy, every page access needs an I/O!

SEQUENTIAL FLOODING

Sequential Flooding: nasty situation caused by LRU policy + repeated sequential scans

- # buffer frames < # pages in file
- each page request causes an I/O !!
- MRU much better in this situation

DBMS vs OS file system

Why not let the OS handle disk management?

- DBMS better at predicting the reference patterns
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool
  - force a page to disk (for recovery & concurrency)
  - adjust the replacement policy
  - pre-fetch pages based on predictable access patterns
- can better control the overlap of I/O with computation
- can leverage multiple disks more effectively

FILE ORGANIZATION

CS 564- Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

File and page organization

- how to organize pages within a file
- how to organize records within a page
- how to organize data within a record
- column stores

MANAGING DISK SPACE

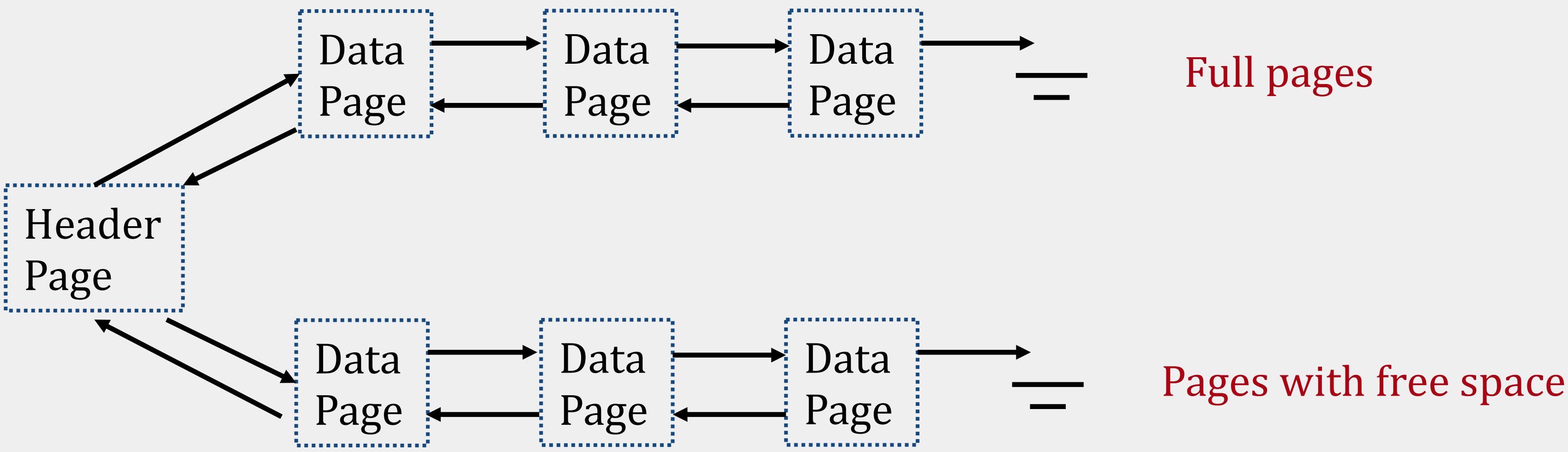
- The disk space is organized into files
- Files are made up of pages
- Pages contain records

UNORDERED (HEAP) FILES

- Contains the records in no particular order
- As file grows/shrinks, disk pages are allocated/deallocated
- To support record level operations, we must keep track of:
  - the pages in a file: page id (*pid*)
  - free space on pages
  - the records on a page: record id (*rid*)

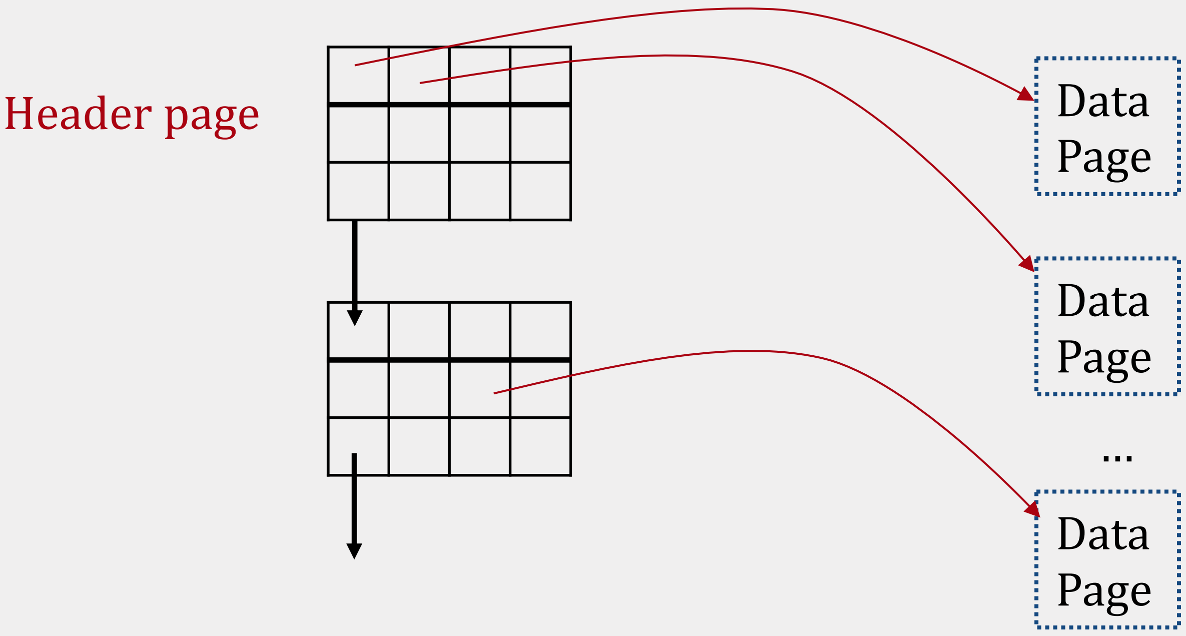
HEAP FILE AS LINKED LIST

- (heap file name, header page id) stored somewhere
- Each page has 2 pointers **+** data
- Pages in the free space list have "some" free space



HEAP FILE AS PAGE DIRECTORY

- Each entry for a page keeps track of:
  - is the page free or full?
  - how many free bytes are?
- We can now locate pages for new tuples faster!



Page Organization

FILES OF RECORDS

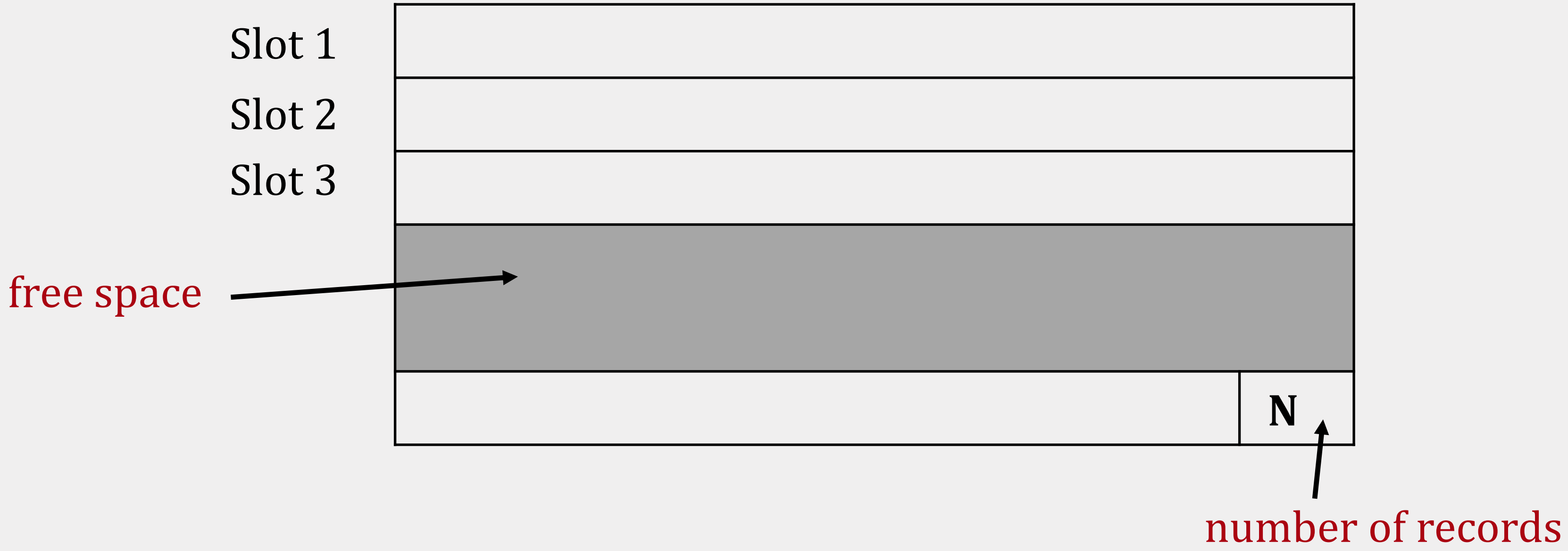
- Page or block is ok for I/O, but higher levels operate on records, and files of records
- File operations:
  - **insert/delete/modify** record
  - **read** a record (specified using the record id)
  - **scan** all records (possibly with some conditions on the records to be retrieved)

PAGE FORMATS

- A page is collection of records
- Slotted page format
  - A page is a collection of slots
  - Each slot contains a record
- ***rid*** *= <page id, slot number>*
- There are many slotted page organizations
- We need to have support for:
  - search, insert, delete records on a page

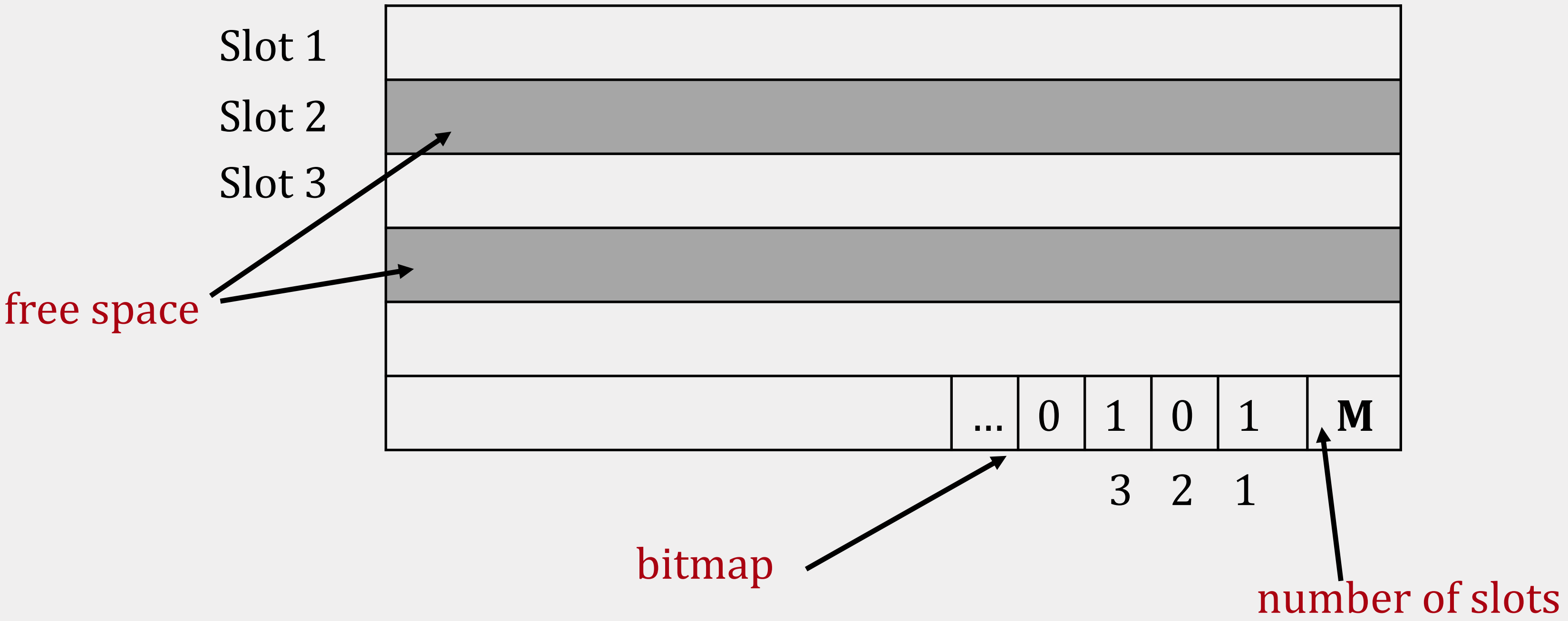
FIXED LENGTH RECORDS (1)

- *packed* organization: N records are always stored in the first N slots
- problem when there are references to records!

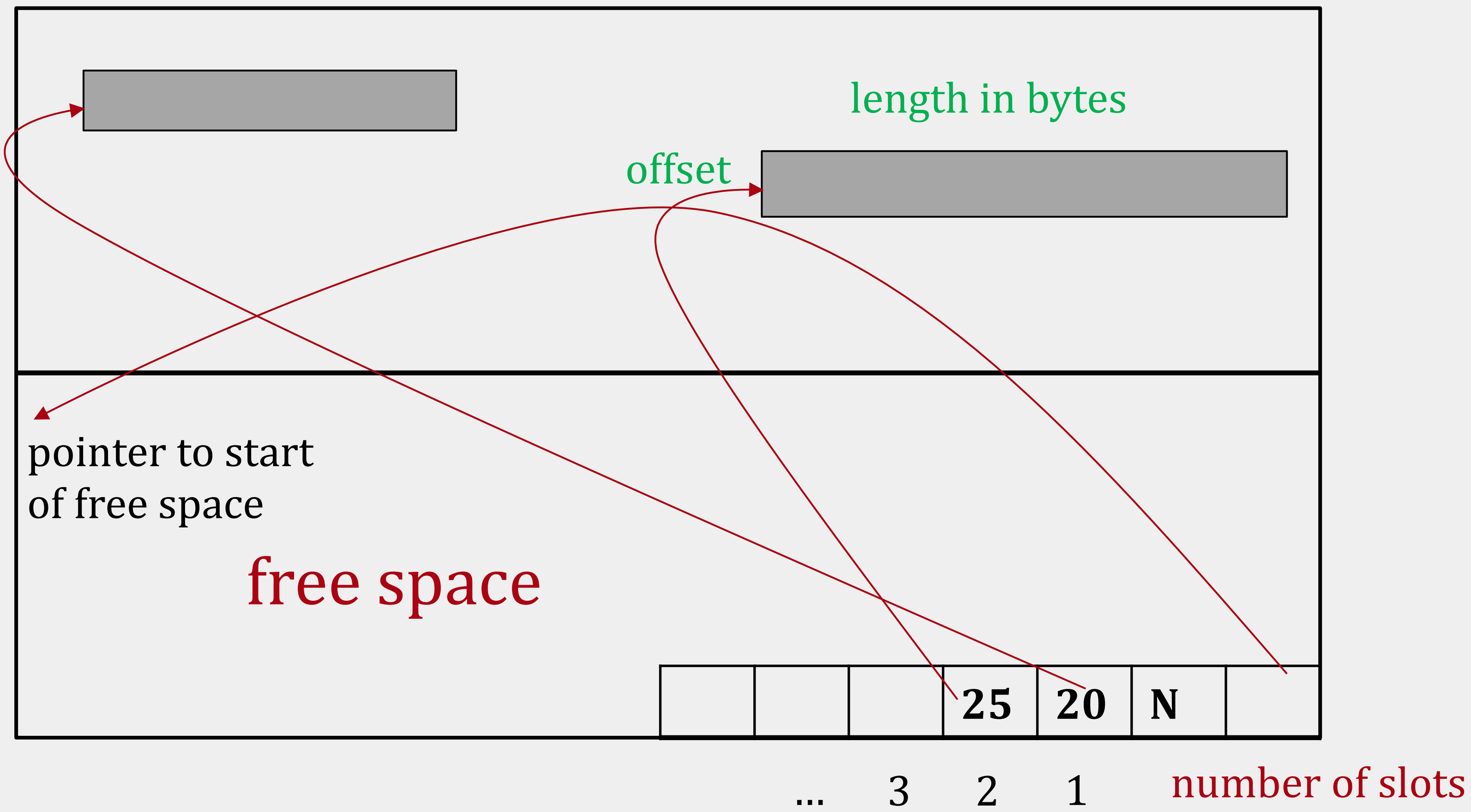


FIXED LENGTH RECORDS (2)

- *unpacked* organization: use a bitmap to locate records in the page



VARIABLE LENGTH RECORDS



VARIABLE LENGTH RECORDS

- Deletion:
  - offset is set to -1
- Insertion:
  - use any available slot
  - if no space is available, reorganize
- *rid* remains unchanged when we move the record (since it is defined by the slot number)

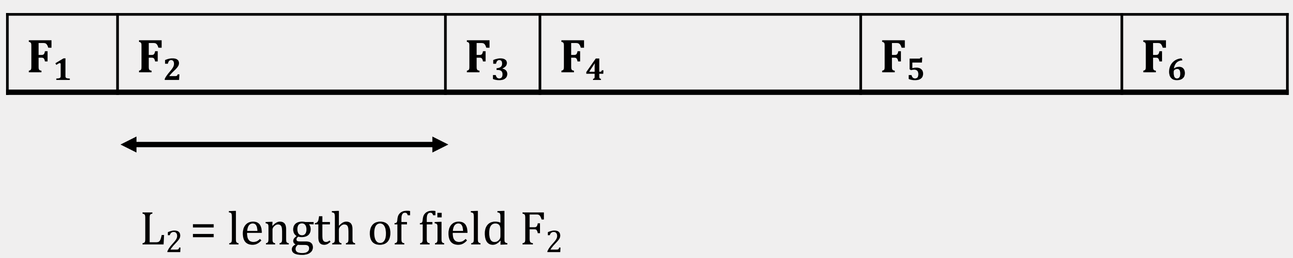
RECORD FORMAT

- How do we organize the field within a record?
  - fixed length
  - variable length

- Information common to all records of a given type is kept in the system catalog:
  - number of fields
  - field type

RECORD FORMAT: FIXED LENGTH

- All records have the same length and same number of fields
- The address of any field can be computed from info in the system catalog!



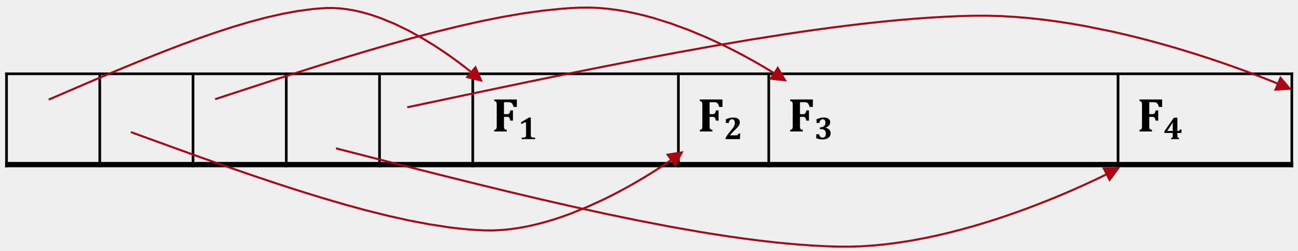
RECORD FORMAT: VARIABLE LENGTH (1)

- store fields consecutively
- use delimiters to denote the end of a field
- need a scan of the whole record to locate a field

F_1	$	F_2	$	F_3	$	F_4	$	F_5

RECORD FORMAT: VARIABLE LENGTH (2)

- store fields consecutively
- use an array of integer offsets in the beginning



BONUS: COLUMN STORES

- Consider a table:
  - **Foo** (a INTEGER, b INTEGER, c VARCHAR(255))
- and the query:
  - `SELECT a FROM Foo WHERE a > 10`
- What could be the problem when we read using the previous record formats?

BONUS: COLUMN STORES

- We can instead store data vertically !
- Each column of a relation is stored in a different file (and can be compressed as well)

row-store

1234	45	Here goes a very long sentence 1
4657	2	Here goes a very long sentence 2
3578	45	Here goes a very long sentence 3

column-store

1234
4657
3578

45
2
45

Here goes a very long sentence 1
Here goes a very long sentence 2
Here goes a very long sentence 3

INDEXING

CS 564- Fall 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

- Indexes
  - alternative file organization
- Index classifications:
  - hash vs tree
  - clustered vs unclustered
  - primary vs secondary

FILE ORGANIZATION: RECAP

- So far we have seen heap files
  - store unordered data
  - fast for scanning all records in a file
  - fast for retrieving by record id (rid)

- But we also need alternative organizations of a file to support other access patterns

MOTIVATION

- Consider the following SQL query:

```
SELECT *
FROM Sales
WHERE Sales.date = “02-11-2016”
```

- For a heap file, we must scan all the pages of the file to return the correct result

ALTERNATIVE FILE ORGANIZATIONS

- We can *speed up* the query execution by better organizing the data in a file
- There are many alternatives:
  - sorted files
  - indexes
    - B+ tree
    - hash index
    - bitmap index

INDEX BASICS

WHAT IS AN INDEX?

- **<u>Index</u>**: a data structure that organizes records of a table to optimize retrieval
  - it speeds up searches for a subset of records, based on values in certain (*search key*) attributes
  - any subset of the attributes can be the search key
  - a search key is *not* the same as the primary key!

- An index contains a collection of *data entries* (each entry with enough info to locate the records)

HASH INDEX: EXAMPLE

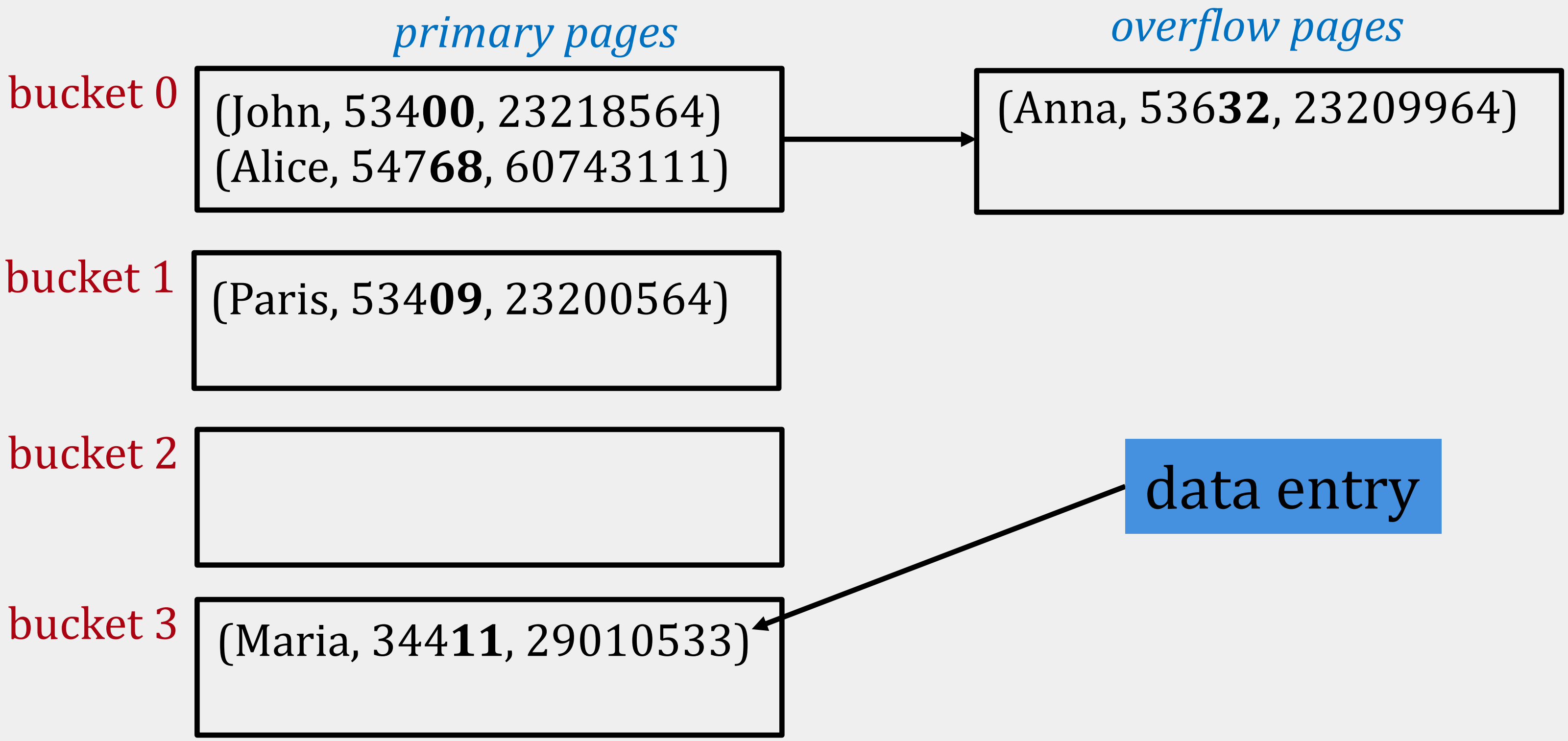
- A **hash index** is a collection of *buckets*
  - bucket = primary page + overflow pages
  - each bucket contains one or more data entries
- To find the bucket for each record, we use a hash function $\boldsymbol{h}$ applied on the search key k
  - $\boldsymbol{N}$ = number of buckets
  - $\boldsymbol{h}(k)\ mod\ \boldsymbol{N}$ = bucket in which the data entry belongs
- Records with different search key may belong in the same bucket!

HASH INDEX: EXAMPLE

`Person(name,zipcode,phone)`

- *search key*: zipcode
- *hash function* ***h***: last 2 digits

- 4 buckets
- each bucket has 2 data entries (full record)



DATA ENTRIES

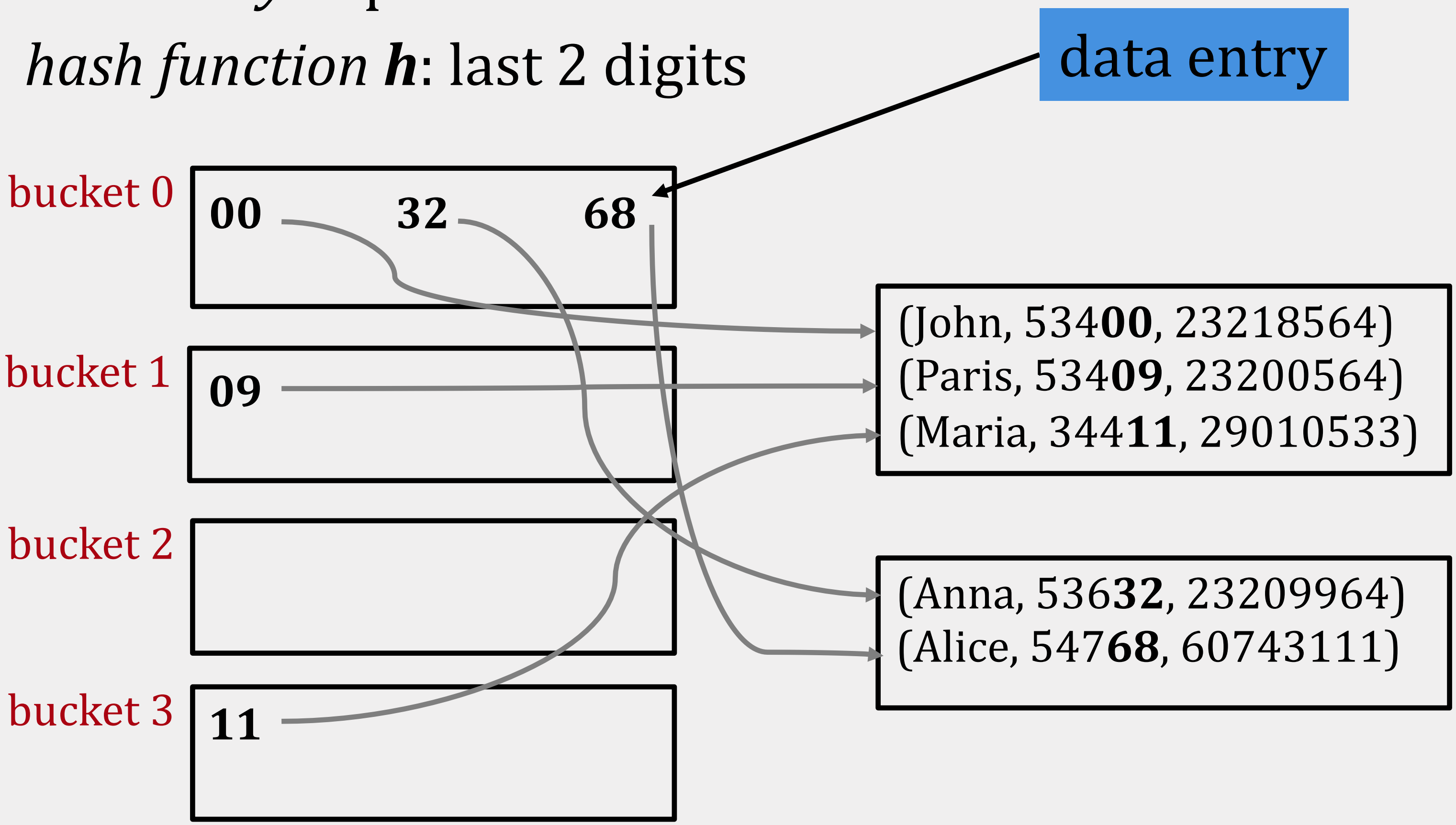
The actual data may not be in the same file as the index!

- In a data entry with search key **k** we have three alternatives of what to store:
  1. the record with key value **k**
  2. <**k**, rid of record with search key value **k**>
  3. <**k**, list of rids of records with search key **k**>
- The choice of alternative for data entries is independent of the indexing technique

EXAMPLE

Person(name,zipcode,phone)

- *search key*: zipcode
- *hash function **h***: last 2 digits



ALTERNATIVES FOR DATA ENTRIES

Alternative #1: *the data entry contains the record*

- the index structure is *by itself* a file organization for records
- *at most one* index on a given collection of data records should use alternative #1
- if data records are very large, the number of pages containing data entries is high
  - this means possibly slower search!

ALTERNATIVES FOR DATA ENTRIES

Alternatives #2, #3: *the data entry contains the rid*

- Data entries are typically much smaller than data records. So, better than #1 with large data records, especially if search keys are small
- #3 is more compact than #2, but leads to variable sized data entries even if search keys are of fixed length

MORE ON INDEXES

A file can have several indexes, on different search keys!

Index classification:

- *primary* **vs** *secondary*
- *clustered* **vs** *unclustered*

PRIMARY VS SECONDARY

- If the search key contains the primary key, it is called a primary index
  - in a primary index, there are no duplicates for a value of the search key
  - there can only be one primary index!
- Any other index is called a secondary index
- If the search key contains a candidate key, it is called a unique index
  - a unique index can also return no duplicates

EXAMPLE

`Sales (sid, product, date, price)`

1. An index on (sid) is a primary and unique index
2. An index on (date) is a secondary, but not unique, index

CLUSTERED INDEXES

Clustered index: the order of records matches the order of data entries in the index

- alternative #1 implies that the index is clustered
- a table can have at most one clustered index
- the cost of retrieving data records through the index varies greatly based on whether index is clustered or not

logical order of index ~ physical order of records

INDEXES IN PRACTICE

CHOOSING INDEXES

- What indexes should we create?
  - which relations should have indexes?
  - what field(s) should be the search key?
  - should we build several or one index?
- For each index, what kind of an index should it be?
  - clustered
  - hash/tree/bitmap

CHOOSING INDEXES

- Consider the best plan using the current indexes, and see if a better plan is possible with an additional index
- One must understand how a DBMS evaluates queries and creates query evaluation plans
- Important trade-offs:
  - queries go faster, updates are slower
  - more disk space is required

CHOOSING INDEXES

- Attributes in `WHERE` clause are candidates for index keys
  - exact match condition suggests hash index
  - indexes also speed up joins (later in class)
  - range query suggests tree index (B+ tree)

- Multi-attribute search keys should be considered when a `WHERE` clause contains several conditions
  - order of attributes is important for range queries
  - such indexes can enable index-only strategies for queries

COMPOSITE INDEXES

Composite search keys: search on a combination of fields (e.g. <date, price>)

- equality query: every field value is equal to a constant value
  - date="02-20-2015" and price =75
- range query: some field value is not a constant
  - date="02-20-2015"
  - date="02-20-2015" and price > 40

INDEXES IN SQL

```
CREATE INDEX index_name
ON table_name (column_name);
```

- Example of simple search key:

```
CREATE INDEX index1
ON Sales (price);
```

INDEXES IN SQL

```
CREATE UNIQUE INDEX index2
ON Sales (sid);
```

- A unique index does not allow any duplicate values to be inserted into the table
- It can be used to check efficiently integrity constraints (a duplicate value will not be allowed to be inserted)

INDEXES IN SQL

```
CREATE INDEX index3
ON Sales (date, price);
```

- Indexes with composite search keys are larger and more expensive to update
- They can be used if we have multiple selection conditions in our queries

Hash Indexes

CS 564- Fall 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

Hash indexes

- Static Hashing
  - what is the I/O cost?
  - problems with static hashing
- Extendible Hashing
  - insertion
  - deletion

HOW TO EVALUATE AN INDEX?

- What access types does it support?
  - *e.g.* equalitiy search, range search, etc.
- Time to access a record
- Time to insert a record
- Time to delete a record
- How much space does it use?

HASH INDEXES

- efficient for equality search
- not appropriate for range search

- Types of hash indexes:
  - static hashing
  - extendible (dynamic) hashing

STATIC HASHING

- A **hash index** is a collection of *buckets*
  - bucket = primary page + overflow pages
  - each bucket contains one or more data entries
- To find the bucket for each record, we use a hash function $\boldsymbol{h}$ applied on the search key k
  - $\boldsymbol{N}$ = number of buckets
  - $\boldsymbol{h}(k)$ *mod* $\boldsymbol{N}$ = bucket in which the data entry belongs
- Records with different search key may belong in the same bucket

STATIC HASHING: EXAMPLE

`Person(name,zipcode,phone)`

- *search key*: zipcode
- *hash function* ***h***: last 2 digits

- 4 buckets
- each bucket has 2 data entries (full record)

	primary pages	*overflow pages*
bucket 0	(John, 534**00**, 23218564) (Alice, 547**68**, 60743111)	(Anna, 536**32**, 23209964)
bucket 1	(Paris, 534**09**, 23200564)	
bucket 2		
bucket 3	(Maria, 344**11**, 29010533)	

OPERATIONS ON HASH INDEXES

Equality search (*search-key = value*)

- apply the hash function on the search key to locate the appropriate bucket
- search through the primary page (plus overflow pages) to find the record(s)

I/O cost = 1 + #overflow pages

OPERATIONS ON HASH INDEXES

- **Deletion**
  - find the appropriate bucket, delete the record
- **Insertion**
  - find the appropriate bucket, insert the record
  - if there is no space, create a new overflow page

HASH FUNCTIONS

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
  - $\boldsymbol{h}(k) = a * k + b$, where a and b are constants
  - a random function

BUCKET OVERFLOW

- Bucket *overflow* can occur because of
  - insufficient number of buckets
  - *skew* in distribution of records
    - many records have the same search key value
    - the hash function results in a non-uniform distribution of key values

- Bucket overflow is handled using *overflow buckets*

PROBLEMS OF STATIC HASHING

- In static hashing, there is a **fixed** number of buckets in the index
- Issues with this:
  - if the database grows, the number of buckets will be too small: long overflow chains degrade performance
  - if the database shrinks, space is wasted
  - reorganizing the index is expensive and can block query execution

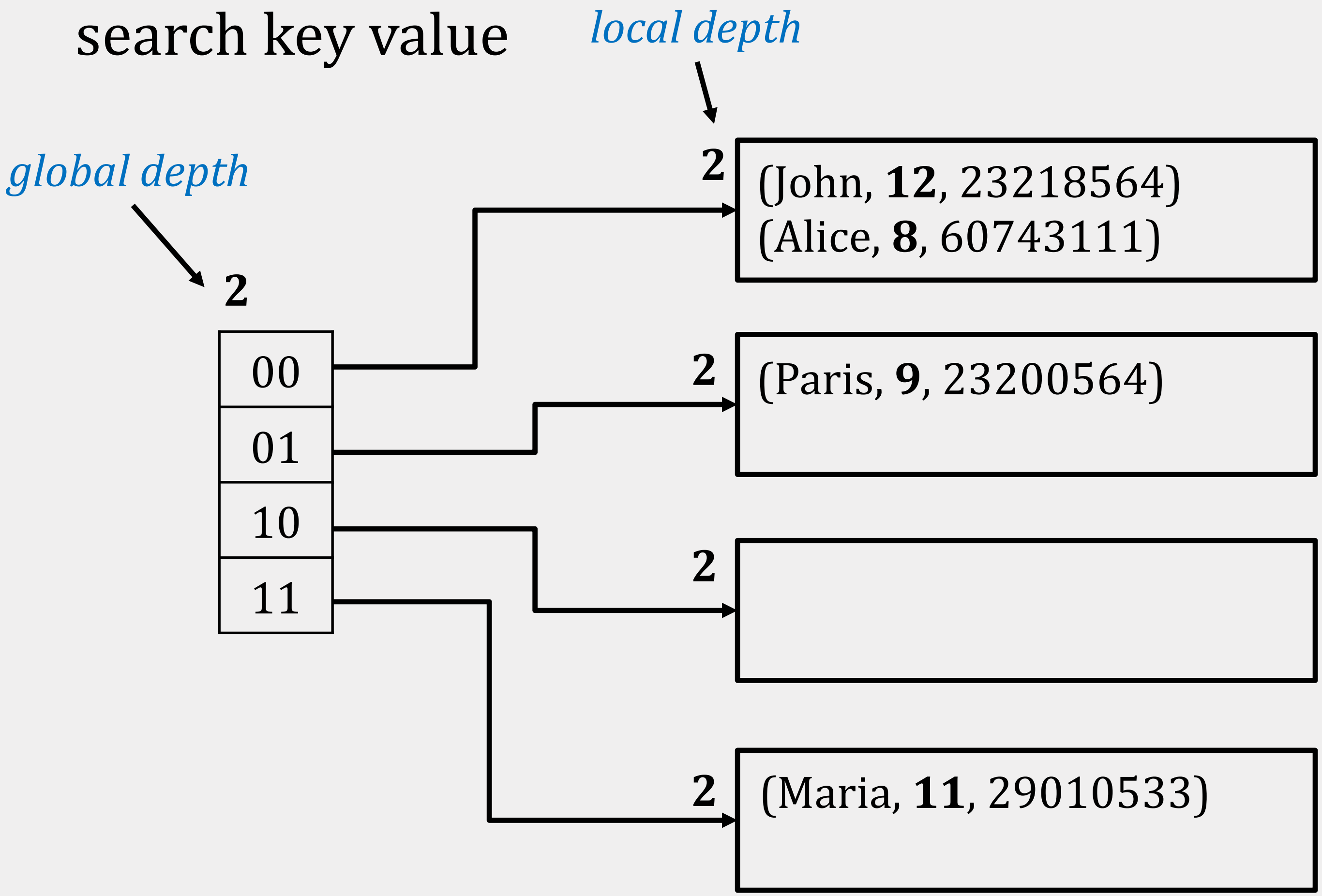
Extendible Hashing

Extendible Hashing

- <u>**Extendible hashing**</u> is a type of *dynamic* hashing
- It keeps a directory of pointers to buckets
- On overflow, it reorganizes the index by doubling the directory (and not the number of buckets)

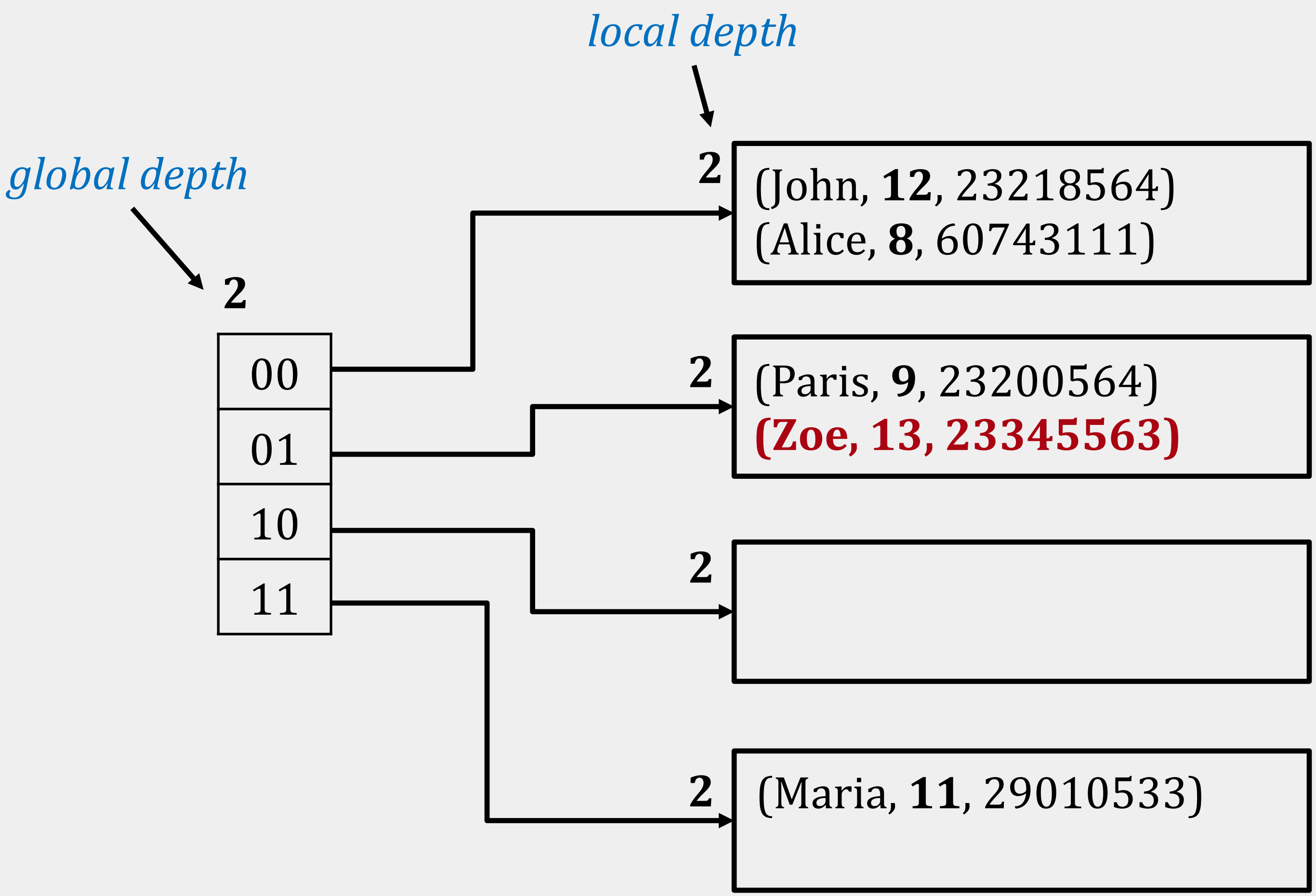
EXTENDIBLE HASHING

To search, use the last **2** digits of the **binary** form of the search key value



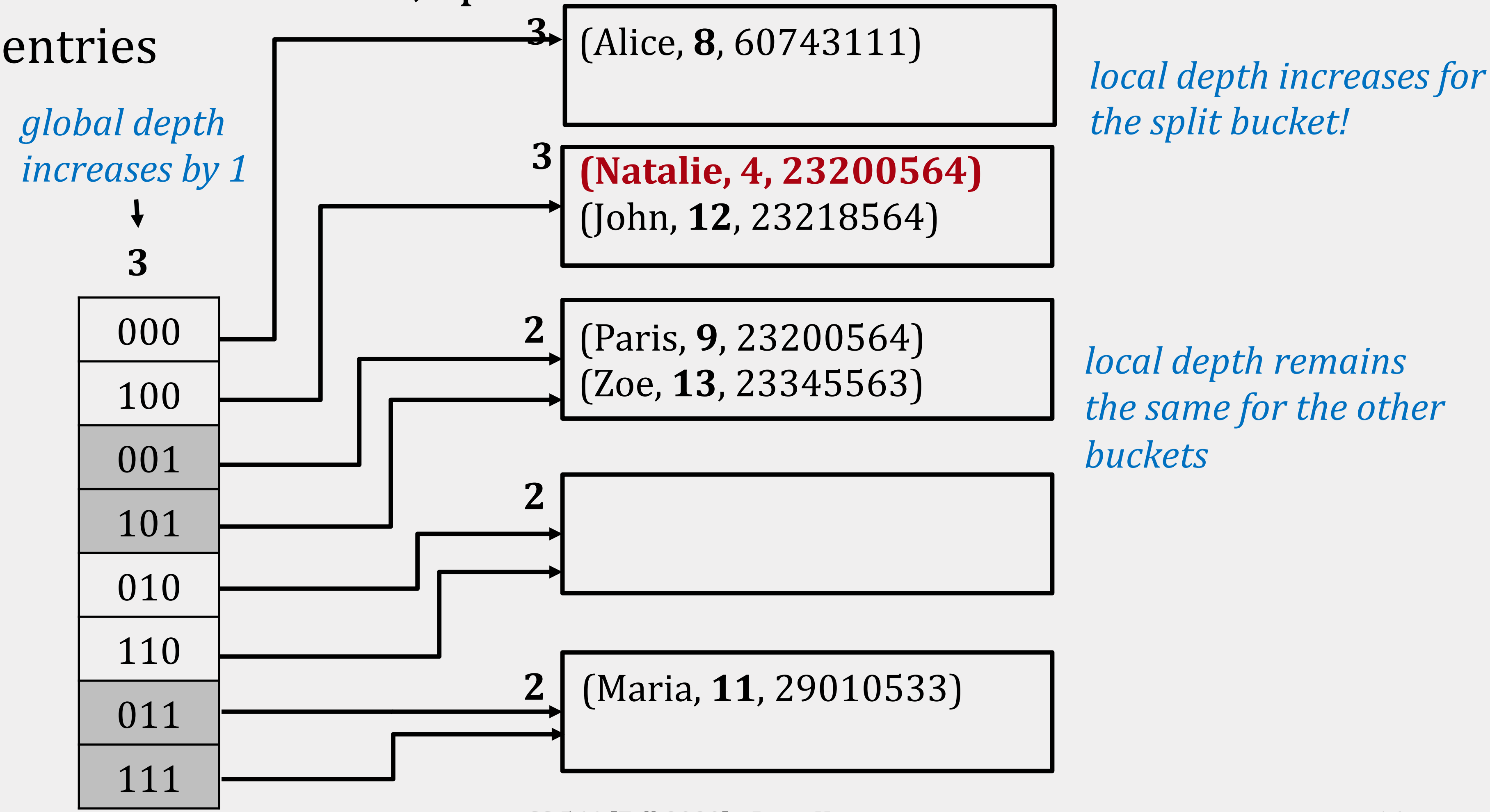
EXTENDIBLE HASHING: INSERT

If there is space in the bucket, simply add the record



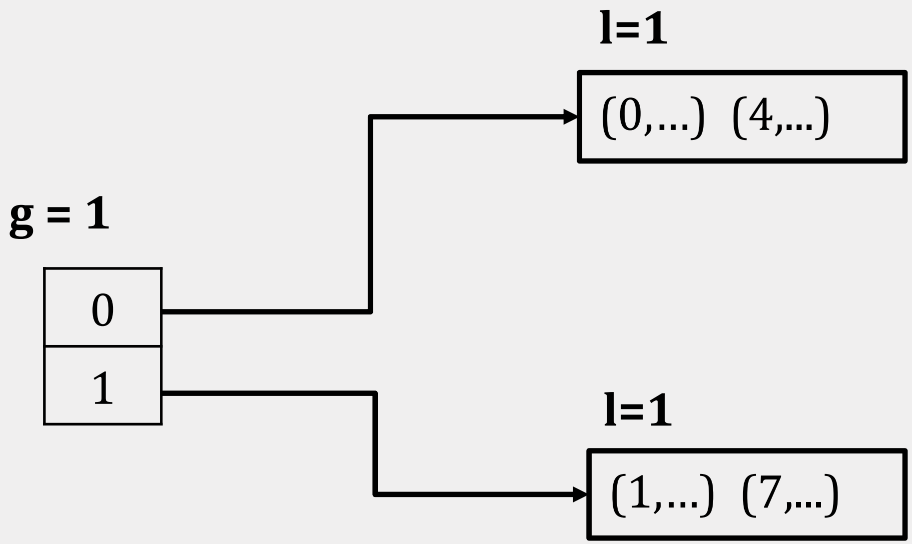
EXTENDIBLE HASHING: INSERT

If the bucket is full, split the bucket and redistribute the entries



EXAMPLE

each page can hold at most two records

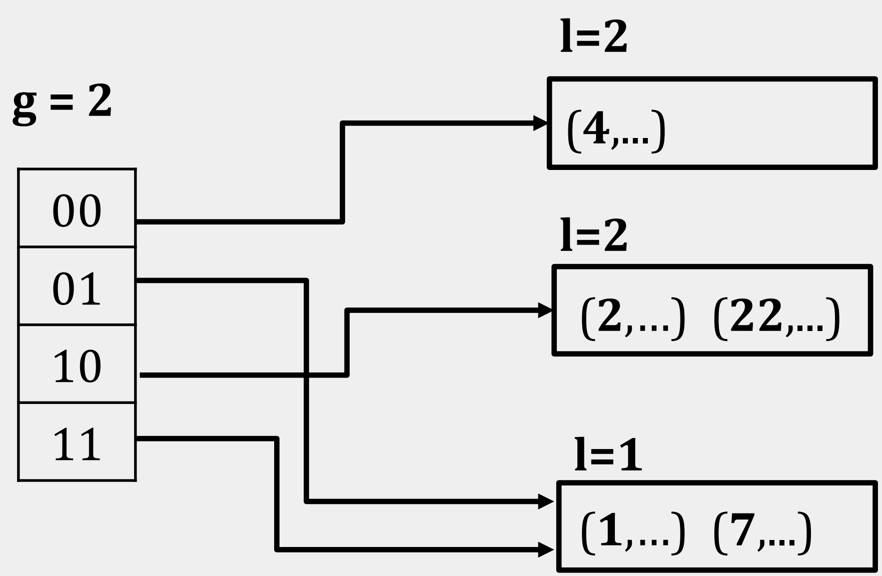


We always have: global depth >= local depth

EXAMPLE

insert: (**22**,…)

- The catalog doubles in size
- Global depth becomes 2



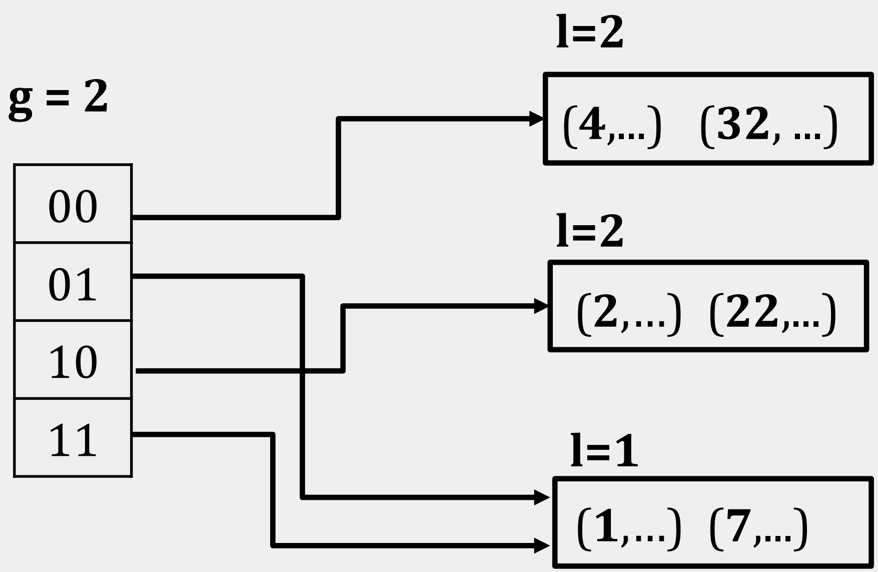
The bucket is split into two buckets with local depth 2

This bucket remains the same

EXAMPLE

insert: (**32**,…)

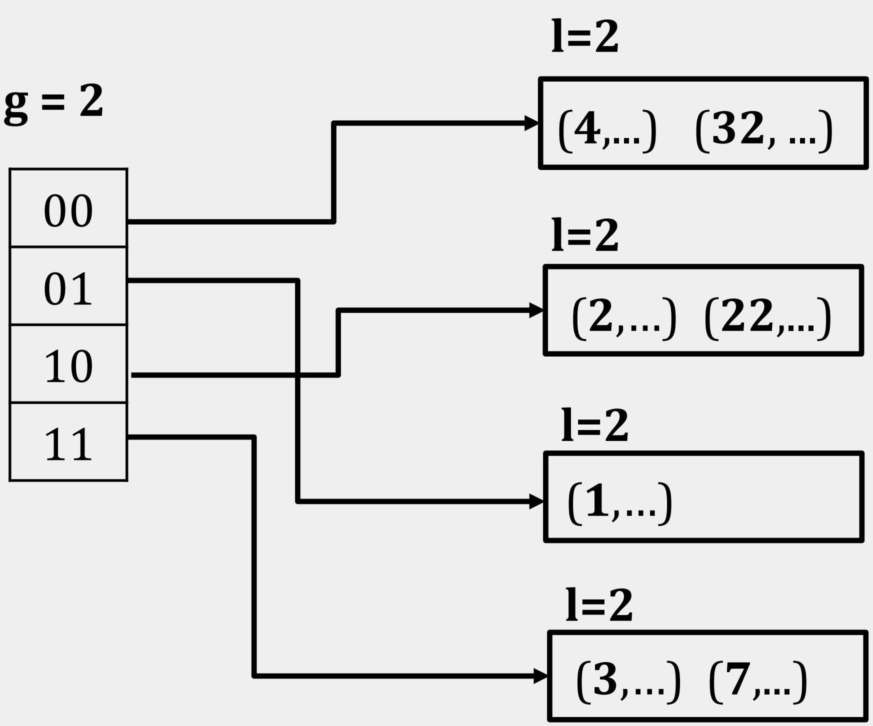
There is space in the bucket so nothing changes!



EXAMPLE

Since local depth is smaller than global, no need to change the directory size!

insert: (**3**,…)



The bucket is split into two

EXTENDIBLE HASHING: DELETE

- Locate the bucket of the record and remove it
- If the bucket becomes empty, it can be removed (and update the directory)
- Two buckets can also be coalesced together if the sum of the entries fit in a single bucket
- Decreasing the size of the directory can also be done, but it is expensive

MORE ON EXTENDIBLE HASHING

- How many disk accesses for equality search?
  - One if directory fits in memory, else two
- Directory grows in spurts, and, if the distribution of hash values is skewed, the directory can grow very large
- We may need overflow pages when multiple entries have the same hash value!

THE B+ TREE INDEX

CS 564- Fall 2020

ACKs: *Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

The **B+ tree** index

- Basics
- Search/Insertion/Deletion
- Design & Cost

INDEX RECAP

- We have the following query:

```
SELECT *
FROM Sales
WHERE price > 100 ;
```

- How do we organize the file to answer this query efficiently?

INDEXES

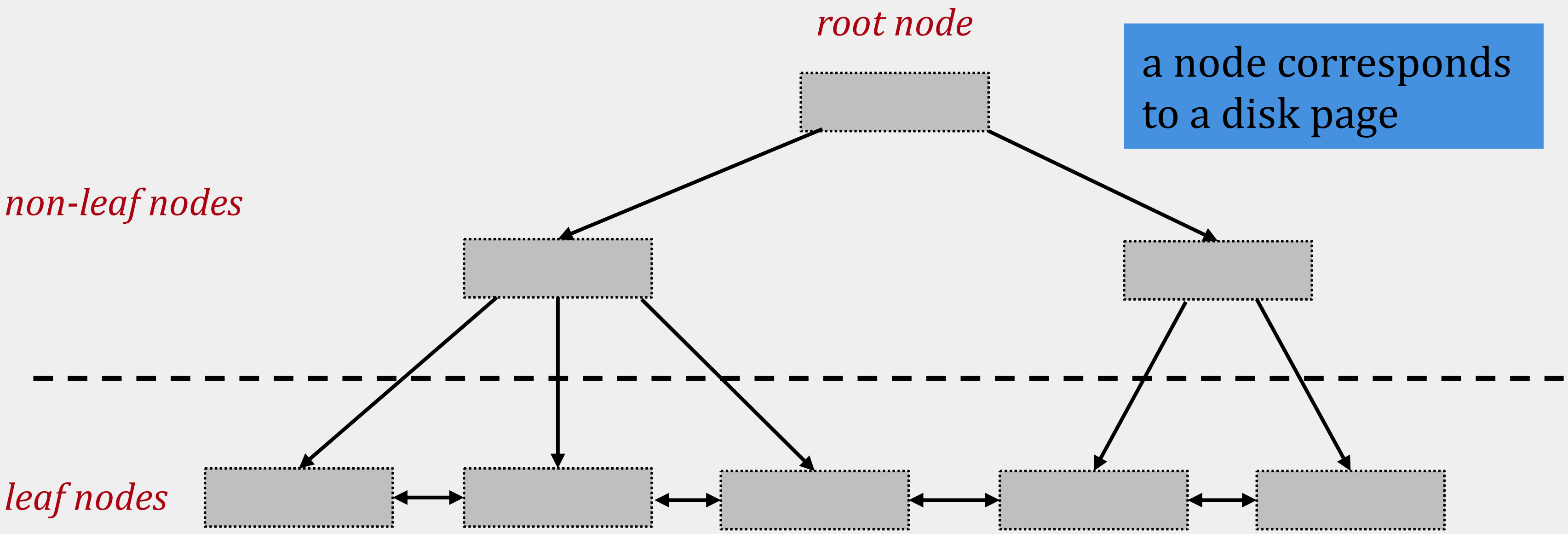
- Hash index:
  - good for equality search
  - in expectation constant I/O cost for search and insert
- B+ tree index:
  - good for **range** and **equality** search

B+ Tree Basics

THE B+ TREE INDEX

- a dynamic tree-structured index
  - adjusted to be always height-balanced
  - 1 node = 1 physical page
- supports efficient equality and range search
- widely used in many DBMSs
  - SQLite uses it as the default index
  - SQL Server, DB2, …

B+ TREE INDEX: BASIC STRUCTURE



data entries

- exist *only* in the leaf nodes
- are sorted according to the search key

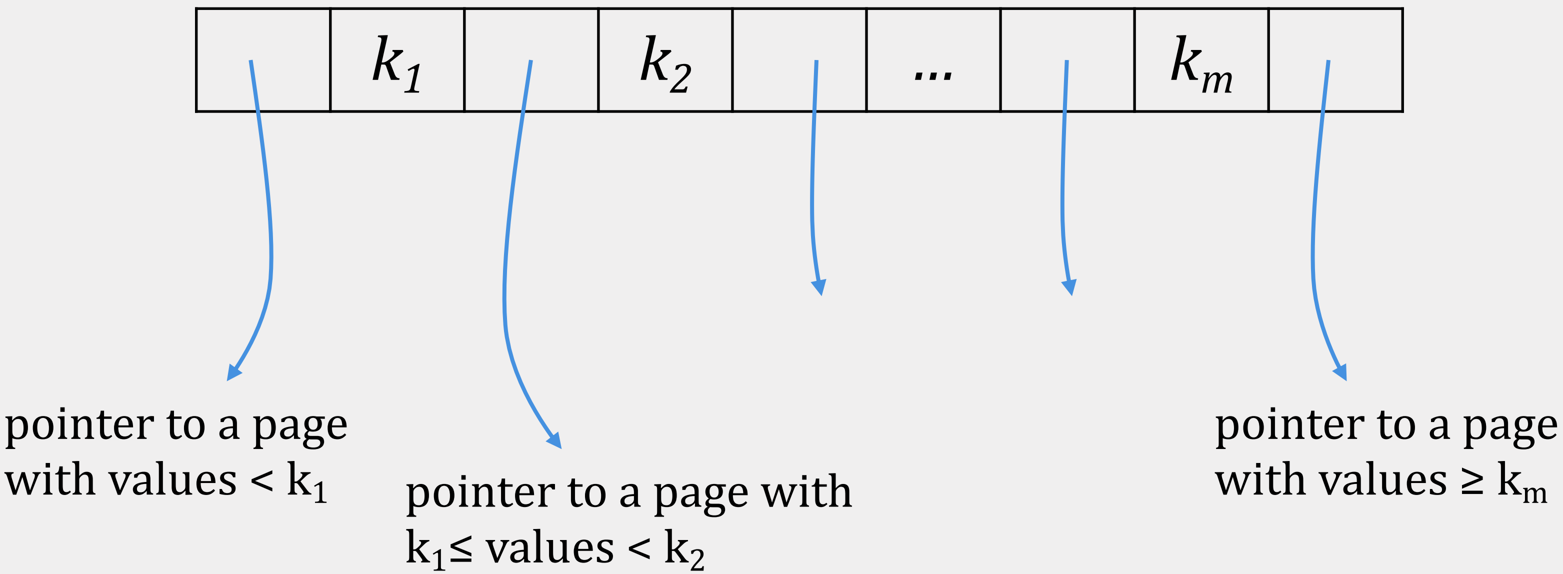
B+ TREE: NODE

- parameter ***d*** is the *order* of the tree
- each node contains $d \leq m \leq 2d$ entries
  - – minimum 50% occupancy at all times
- with the exception of the root node, which can have $1 \leq m \leq 2d$ entries

	k_1		k_2		…		k_m	

NON-LEAF NODES

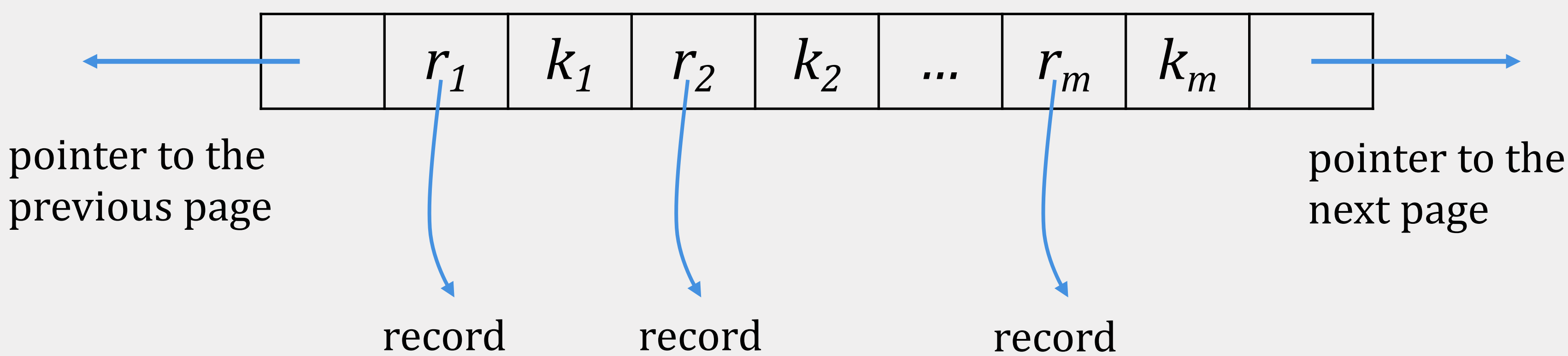
A non-leaf (or internal) node with *m* entries has *m+1* pointers to lower-level nodes



LEAF NODES

A leaf node with m entries has

- m pointers to the data records (rids)
- pointers to the next and previous leaves



B+ Tree Operations

B+ TREE OPERATIONS

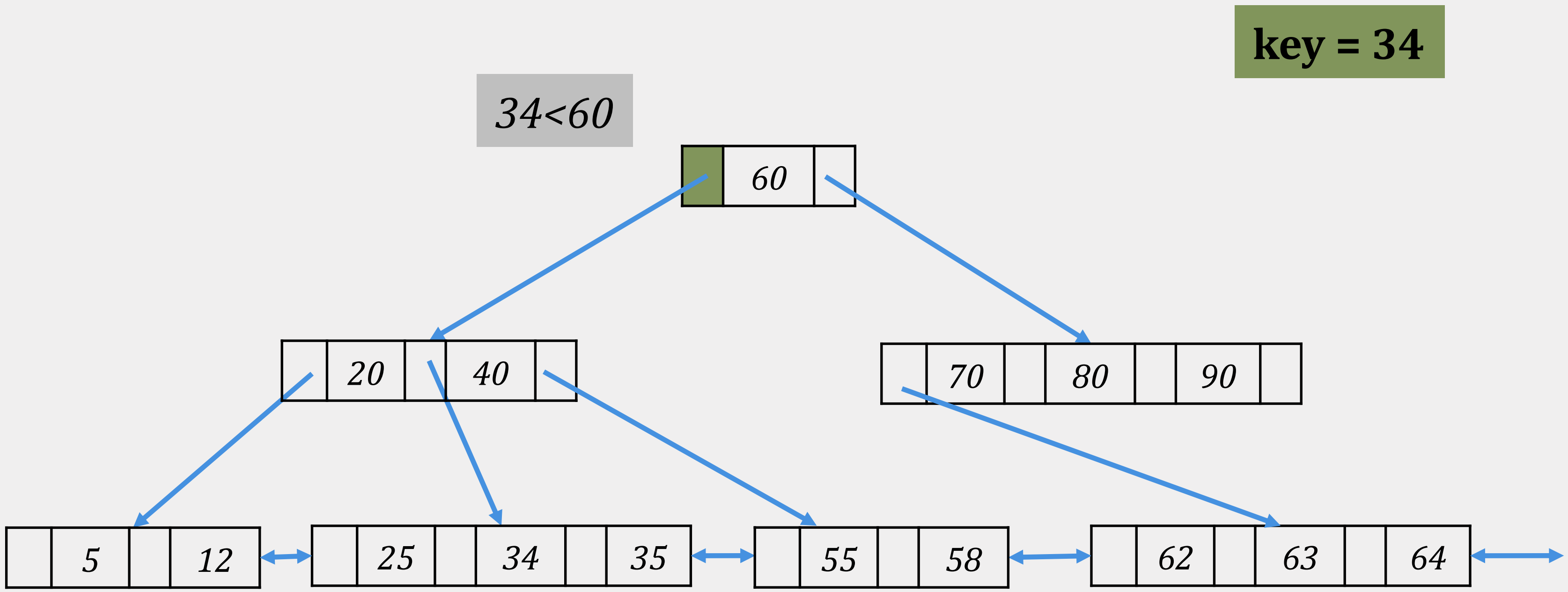
A B+ tree supports the following operations:

- equality search
- range search
- insert
- delete
- bulk loading

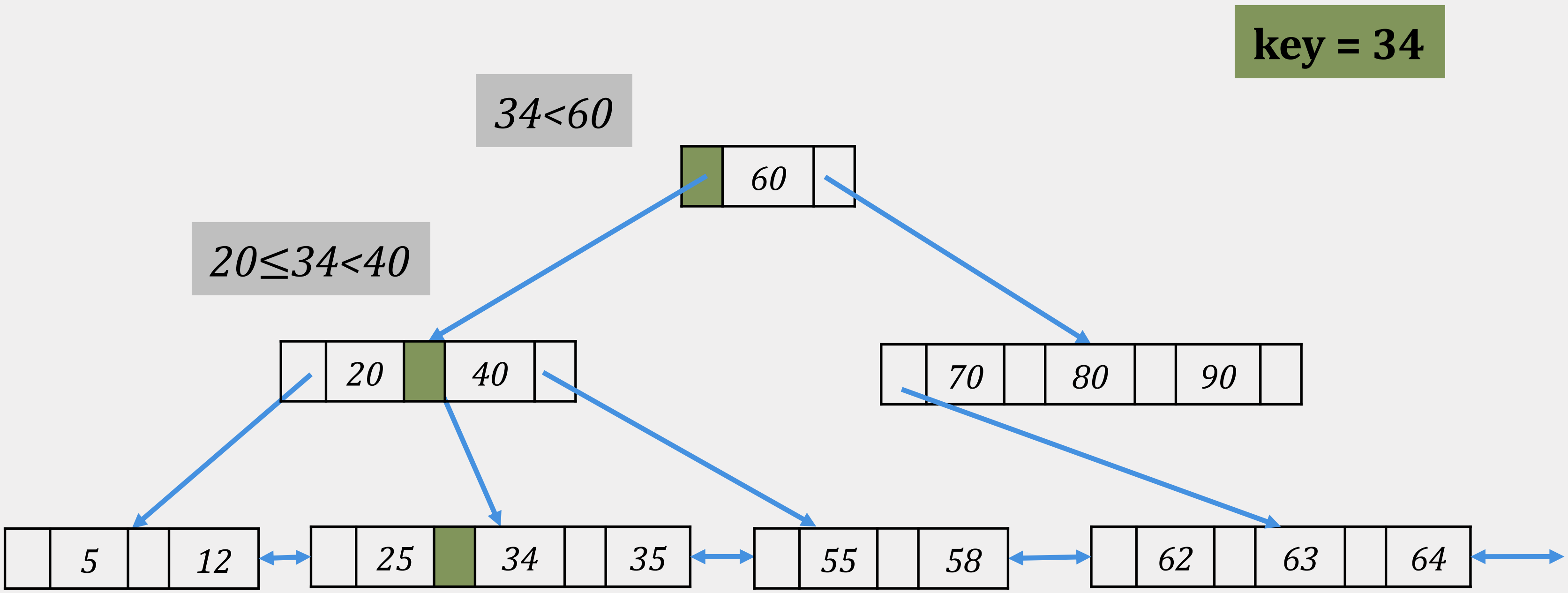
SEARCH

- start from the root node
- examine the index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
  - for equality search, we are done
  - for range search, traverse the leaves sequentially using the previous/next pointers

EQUALITY SEARCH: EXAMPLE

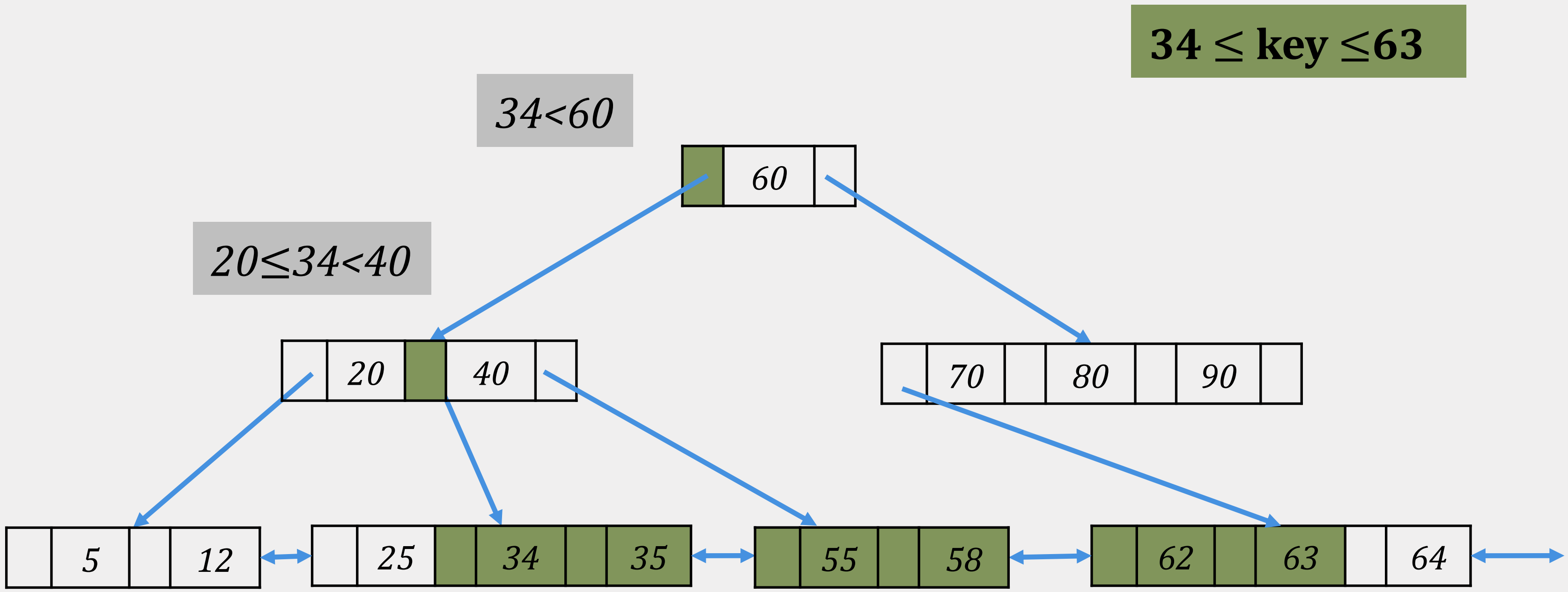


EQUALITY SEARCH: EXAMPLE



To locate the correct data entry in the leaf node, we can do either linear or binary search

RANGE SEARCH: EXAMPLE



After we find the leftmost point of the range, we traverse sequentially!

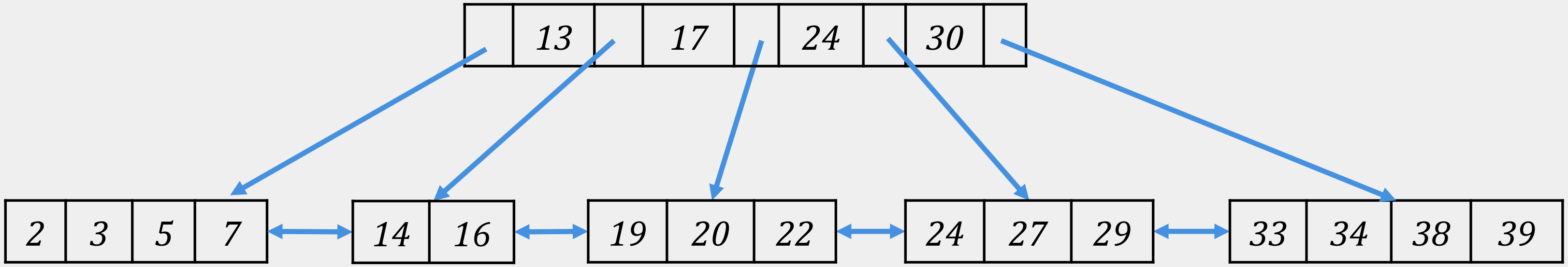
INSERT

- find correct leaf node **L**
- insert data entry in **L**
  - If **L** has enough space, DONE!
  - Else, we must split **L** (into **L** and a new node **L'**)
    - redistribute entries evenly, **copy up** the middle key
    - insert index entry pointing to **L'** into parent of **L**
- This can propagate recursively to other nodes!
  - to split a non-leaf node, redistribute entries evenly, but **push up** the middle key

INSERT: EXAMPLE

order **d** = 2

insert 8

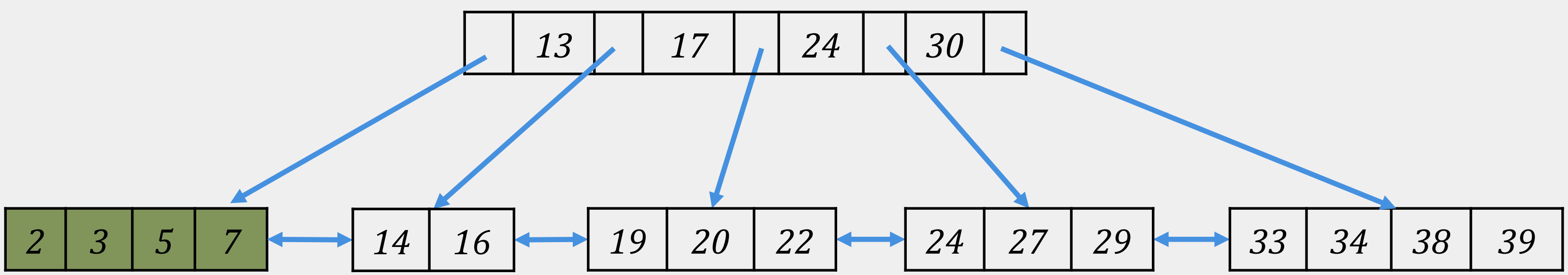


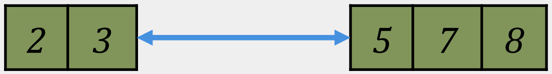
INSERT: EXAMPLE

order **d** = 2

insert 8

the leaf node is full so we must split it!





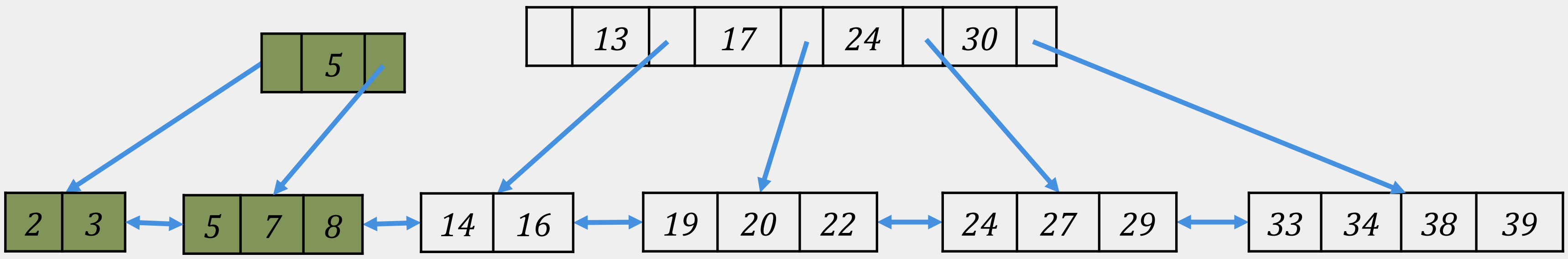
d entries d+1 entries

INSERT: EXAMPLE

order **d** = 2

insert 8

the middle key (***5***) must be copied up, but the root node is full as well!

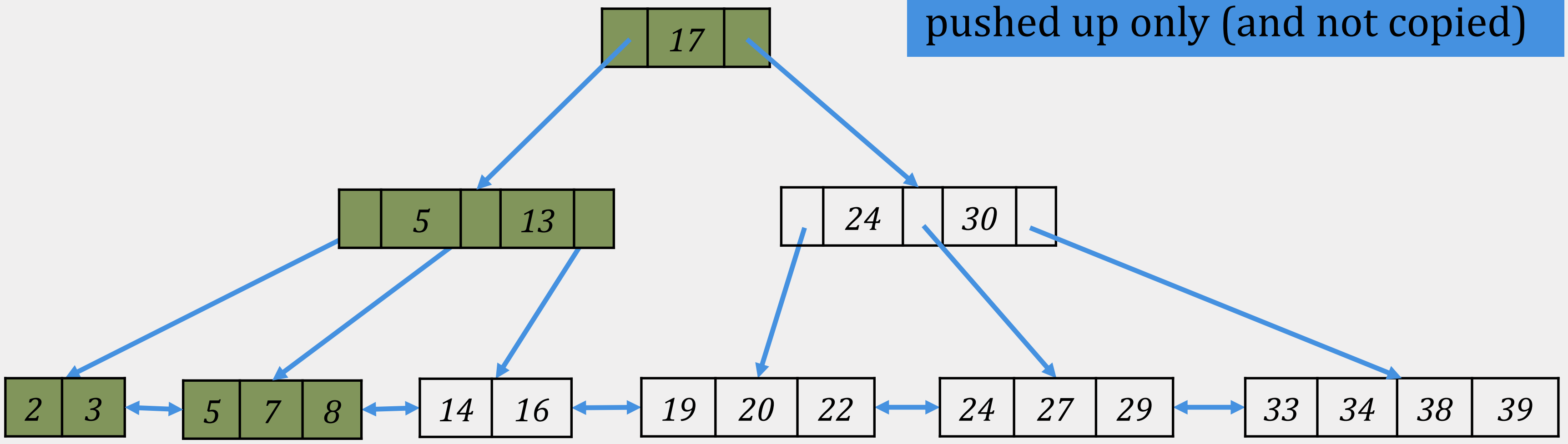


INSERT: EXAMPLE

order **d** = 2

insert 8

the new middle key (***17***) is now pushed up only (and not copied)



INSERT PROPERTIES

The B+ Tree insertion algorithm has several attractive qualities:

- ~ same cost as exact search
- it is ***self-balancing:*** the tree remains balanced (with respect to height) even after multiple insertions

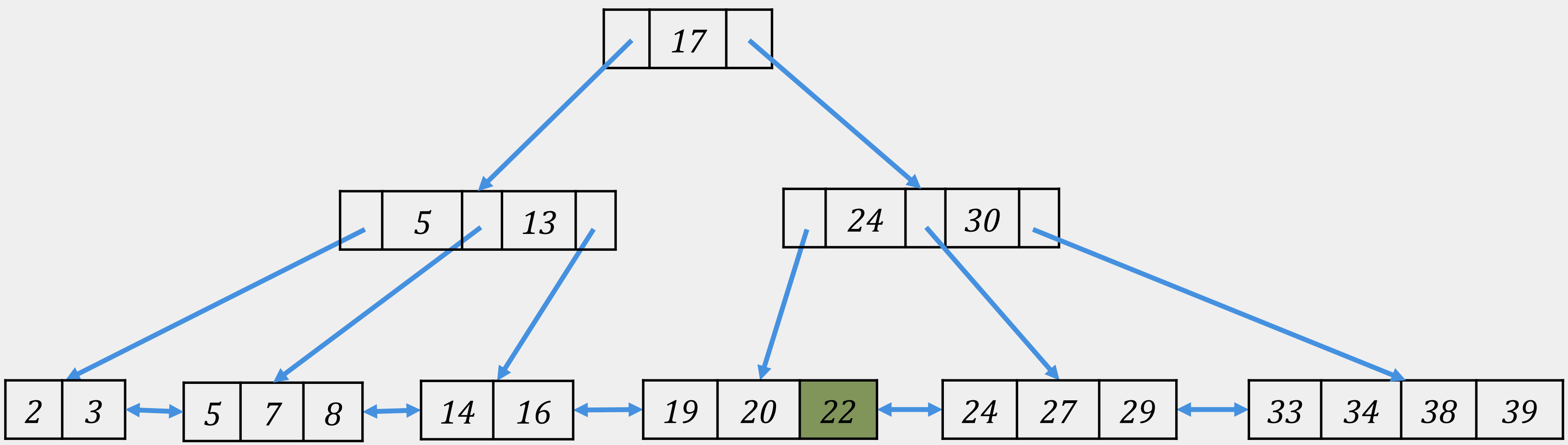
B+ TREE: DELETE

- find leaf node **L** where entry belongs
- remove the entry
  - If **L** is at least half-full, DONE!
  - If **L** has only *d-1* entries,
    - Try to **re-distribute**, borrowing from sibling
    - If re-distribution fails, **merge L** and sibling
- If a merge occurred, we must delete an entry from the parent of **L**

DELETE : EXAMPLE 1

order **d** = 2

delete 22

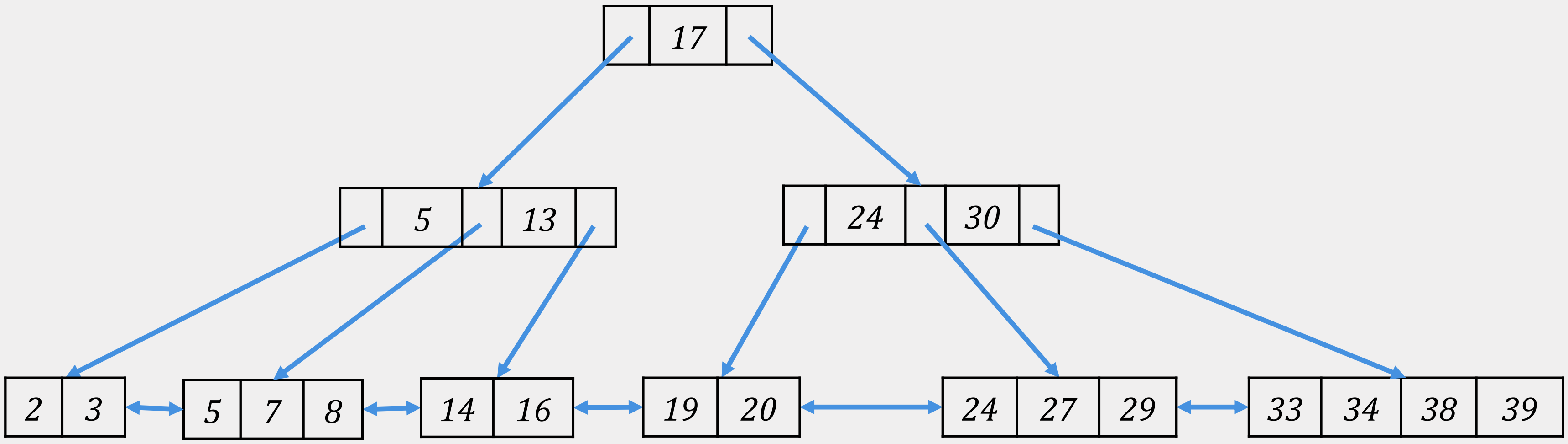


since by deleting 22 the node remains half-full, we simply remove it

DELETE : EXAMPLE 1

order **d** = 2

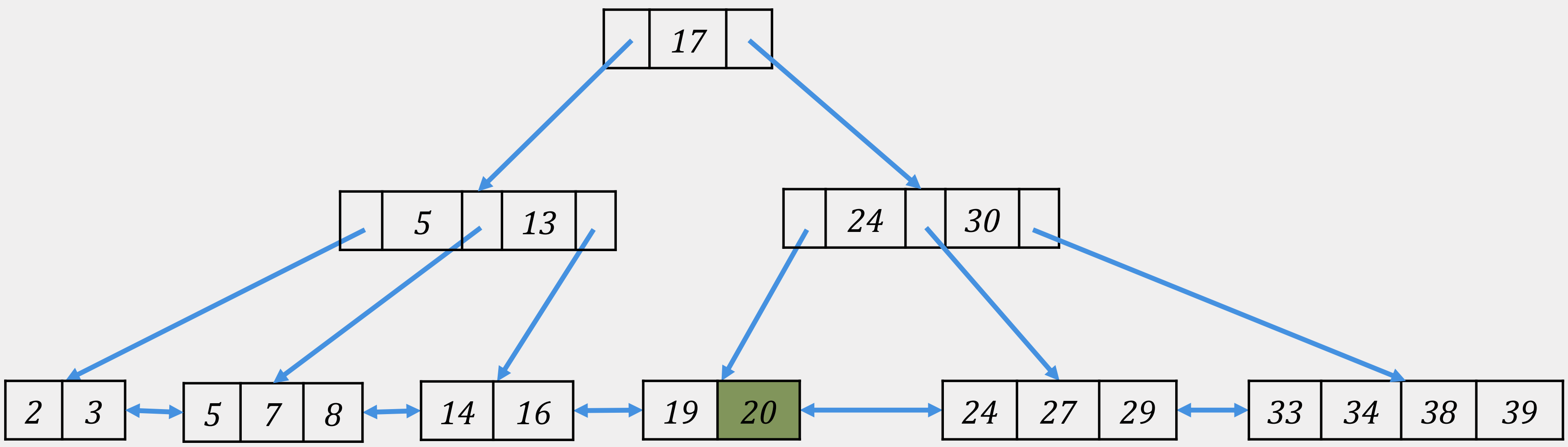
delete 22



DELETE : EXAMPLE 2

order **d** = 2

delete 20

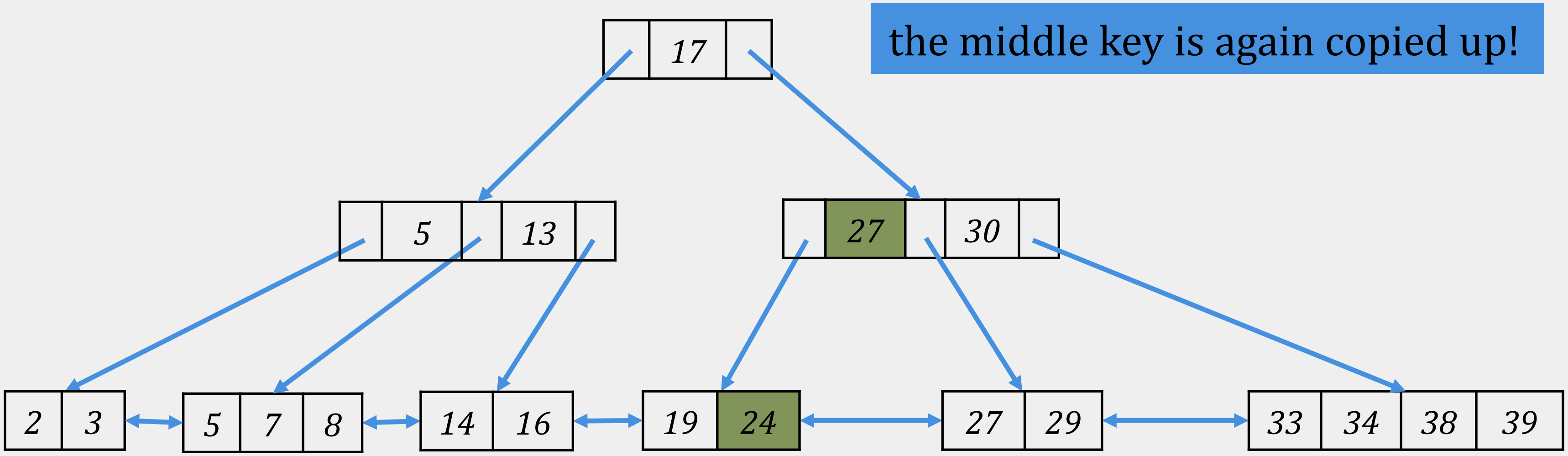


by removing 20 the node is not half-full anymore, so we attempt to redistribute!

DELETE : EXAMPLE 2

order **d** = 2

delete 20



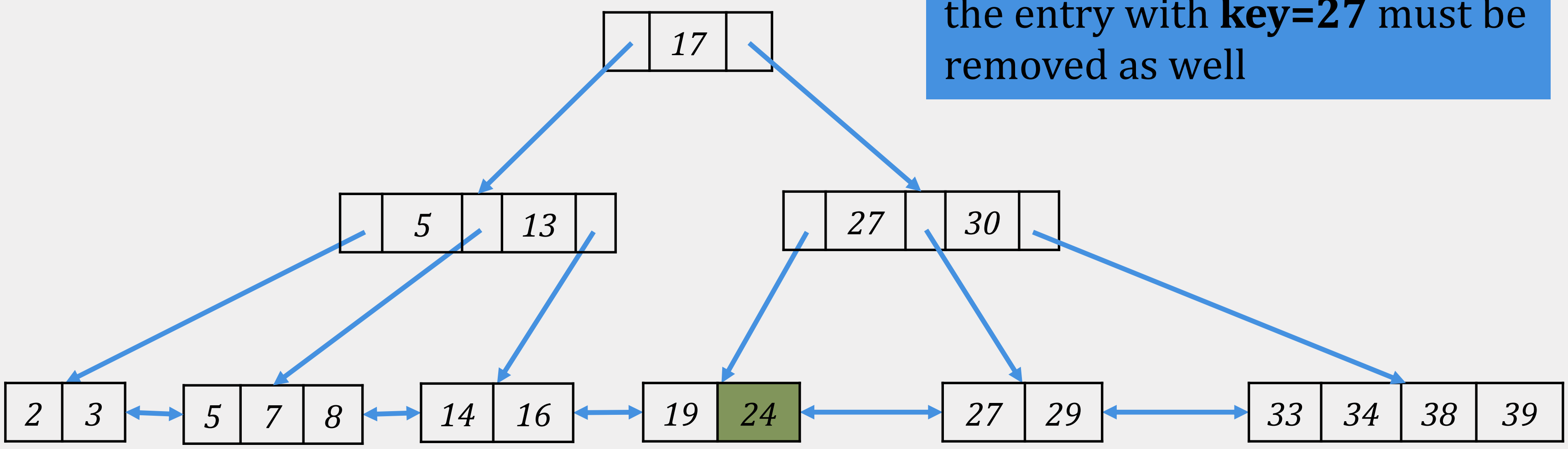
the middle key is again copied up!

by removing 20 the node is not half-full anymore, so we attempt to redistribute!

DELETE : EXAMPLE 3

order **d** = 2

delete 24



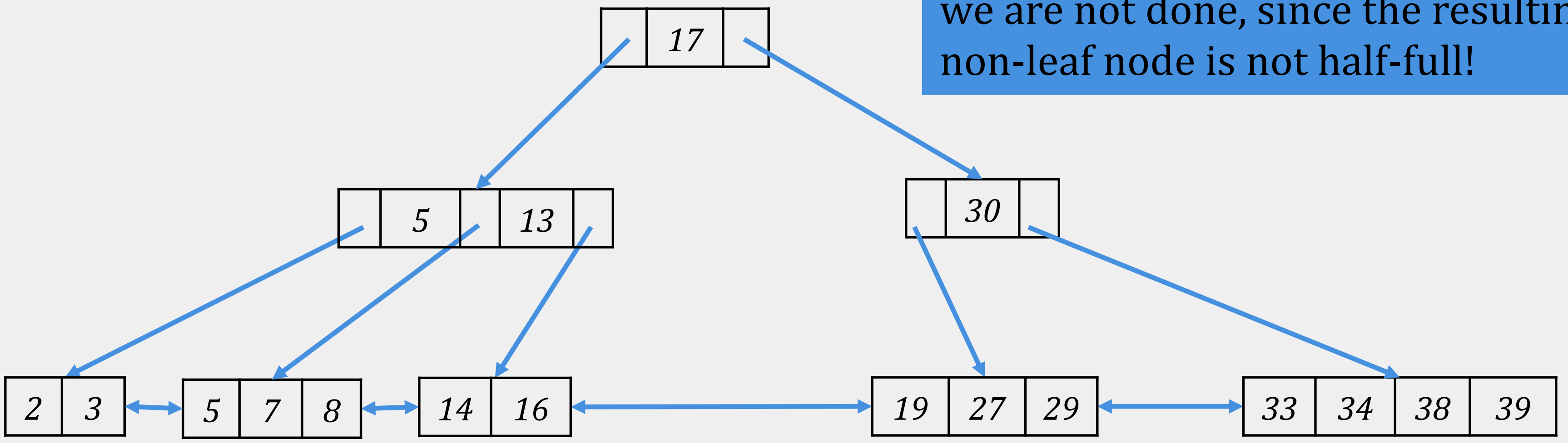
the entry with **key=27** must be removed as well

in this case, we have to merge nodes!

DELETE : EXAMPLE 3

order **d** = 2

delete 24



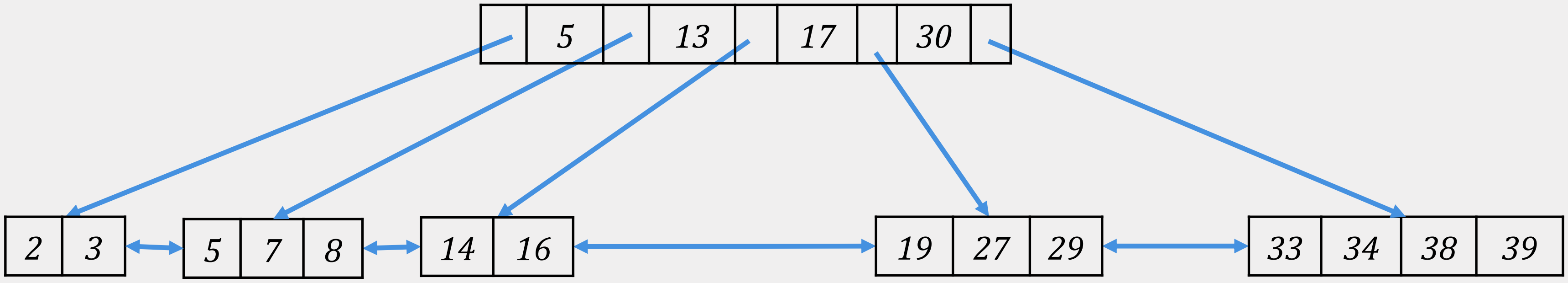
we are not done, since the resulting non-leaf node is not half-full!

DELETE : EXAMPLE 3

order **d** = 2

delete 24

we are not done, since the resulting non-leaf node is not half-full!



B+ TREE: DELETE

- Redistribution of entries can also be possible for the non-leaf nodes
- We can also try to redistribute using *all siblings*, and not only the neighboring one

DUPLICATES

- **duplicate keys**: many data entries with the same key value
- Solution 1:
  - All entries with a given key value reside on a single page
  - Use overflow pages
- Solution 2:
  - Allow duplicate key values in data entries
  - Modify search operation

B+ Tree Design & Cost

B+ TREE: FAN-OUT

fan-out f: the number of pointers to child nodes coming out of a non-leaf node

- compared to binary trees (fan-out =2), B+ trees have a high fan-out ($d+1 \leq f \leq 2d+1$)
- The fan-out of B+ trees is dynamic, but we will often assume it is constant for our cost model

B+ TREE: FILL-FACTOR

fill-factor ***F***: the percent of available slots in the B+ Tree that are filled

- it is usually < 1 to leave slack for (quicker) insertions!
- typical fill factor F = 2/3

B+ TREE: HEIGHT

height ***h***: the number of levels of the non-leaf nodes

- the height is at least 1 (root node)
- high fan-out -> smaller height-> less I/O per search
- typical heights of B+ trees: 3 or 4

B+ TREE: EXAMPLE

- page size **P** = 4000 bytes
- search key size = 30 bytes
- address size = 10 bytes
- fill-factor **F** = 2/3
- number of records = 2,000,000

- We assume that the data entries store only the search key and the address of tuple
- We assume no duplicate entries

B+ TREE: EXAMPLE

What is the order **d** and fan-out **f** ?

- each non-leaf node stores up to 2**d** values of the key + (2**d**+1) addresses for the children pages
- to fit this into a single page, we must have:

$$2d \cdot 30 \ + (2d+1) \cdot 10 \ \leq \ 4000$$

$$\boldsymbol{d} \leq 50$$

- since a maximum capacity node has (2**d**+1) = 101 children, and the fill-factor is 2/3, the fan-out is $\boldsymbol{f} = 101 \ * \frac{2}{3} = 67$

B+ TREE: EXAMPLE

How many leaf pages are in the B+ tree?

- we assume for simplicity that each leaf page stores only pairs of (key, address)
- each pair needs 30+10 = 40 bytes
- to store 2,000,000 such pairs with fill-factor **F** = 2/3, we need:

$$\#leaves \; = (2{,}000{,}000 * 40)/(4{,}000 * \boldsymbol{F}) = \; 30{,}000$$

B+ TREE: EXAMPLE

What is the height **h** of the B+ tree?

- we calculated that we need to index **N** = 30,000 pages
- **h** = 1 -> indexes **f** pages
- **h** = 2 -> indexes $\mathbf{f}^2$ pages
- …
- **h** = k -> indexes $\mathbf{f}^k$ pages

height must be $\boldsymbol{h} = \lceil log_f N \rceil$

for our example, $\boldsymbol{h} = \lceil log_{67} 30{,}000 \rceil = 3$

B+ TREE: EXAMPLE

What is the total size of the tree?

- #pages = 1 + 67 + 67^2 + 30,000 = 34,557
- the top levels of the B+ tree do not take much space and can be kept in the buffer pool
  - *level 0* = 1 page ~ 4 KB
  - *level 1* = 67 pages ~ 268 KB
  - *level 2* = 4,489 pages ~ 18 MB

COST MODEL FOR SEARCH

To do equality search:

- we read one page per level of the tree
- levels that we can fit in buffer are free!
- finally we read in the actual record

I = 0 if the record is stored at the leaf node, otherwise I = 1

$$\text{I/O cost } = \boldsymbol{h} - L_B + 1 + I$$

If we have $\boldsymbol{B}$ available buffer pages, we can store $\boldsymbol{L}_{\mathrm{B}}$ levels of the B+ Tree in memory:

- $\boldsymbol{L}_{\mathrm{B}}$ is the number of levels such that the sum of all the levels' nodes fit in the buffer:
$$B \geq 1 + f + \cdots + f^{L_B - 1}$$

COST MODEL FOR SEARCH

To do range search:

- we read one page per level of the tree
- levels that we can fit in buffer are free!
- we read sequentially the pages in the range

$$\text{I/O cost } = \text{h} - L_B + OUT$$

Here, *OUT* is the I/O cost of loading the additional leaf nodes we need to access + the I/O cost of loading each ***page*** of the results

Bitmap Indexes

CS 564- Spring 2020

ACKs: *Jignesh Patel, Theo Rekatsinas*

WHAT IS THIS LECTURE ABOUT?

- Bitmap Indexes
- Bitslice Indexes

MOTIVATION

Consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,        -- unique message id
    tstamp      TIMESTAMP,      -- when was the tweet posted
    uid         INTEGER,        -- unique id of the user
    msg         VARCHAR (140),  -- the actual message
    zip          INTEGER,        -- zipcode when posted
    retweet      BOOLEAN        -- retweeted?
    );
```

How can we speed up the following query?

```
SELECT * FROM Tweets
WHERE zip BETWEEN 53000 AND 54999 ;
```

B+ tree on attribute zip

MOTIVATION

Consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,        -- unique message id
    tstamp      TIMESTAMP,      -- when was the tweet posted
    uid         INTEGER,        -- unique id of the user
    msg         VARCHAR (140),  -- the actual message
    zip         INTEGER,        -- zipcode when posted
    retweet     BOOLEAN         -- retweeted?
    );
```

How many bytes does a B+ tree use for each record?

- at least key + rid, so: key-size + rid-size

Can we do better than that (in terms of storage overhead)?

- **yes!** especially when the attribute domain is small

BITMAP INDEX

THE BITMAP INDEX

- Consider building an index to answer *equality* queries on the retweet attribute
- Issues with building a B+ tree:
  - three distinct values: `yes`, `no`, `NULL`
  - many duplicates for each distinct value
  - a weird B+ tree with three long rid lists
- **bitmap index**: build three *bitmap arrays* (stored on disk), one for each value
  - the i^{th} bit in each bitmap corresponds to the i^{th} tuple

    (we need to map the i^{th} position to a rid!)

BITMAP: EXAMPLE

table (stored in heapfile)

uniqueMsgID	…	zip	retweet
1	…	11324	yes
2	…	53705	yes
3	…	53706	no
4	…	53705	NULL
5	…	90210	no
…	…	…	…
1,000,000,000	…	53705	yes

bitmap index (on retweet)

yes
1
1
0
0
0
…
1

no
0
0
1
0
1
…
0

null
0
0
0
1
0
…
0

```
SELECT * FROM Tweets WHERE retweet = “no” ;
```

- scan the “no” bitmap file
- for each bit set to 1, compute the tuple rid
- fetch the tuple

A CRITICAL ISSUE

- We need an efficient way to compute a bit position:
  - layout the bitmap in page-id order
- We need an efficient way to map a bit position to a rid:
  - fix the # records per page in the heapfile
  - lay the pages out so that page-ids are sequential and increasing
  - then construct rid (page-id, slot#)
    - page-id = bit-position / #records-per-page
    - slot# = bit-position % #records-per-page

With variable length records, we have to set the limit based on the size of the largest record, which may result in under-filled pages!

BITMAP: OTHER QUERIES

table (stored in heapfile)

uniqueMsgID	…	zip	retweet
1	…	11324	yes
2	…	53705	yes
3	…	53706	no
4	…	53705	NULL
5	…	90210	no
…	…	…	…
1,000,000,000	…	53705	yes

bitmap index (on retweet)

yes
1
1
0
0
0
…
1

no
0
0
1
0
1
…
0

null
0
0
0
1
0
…
0

```
SELECT COUNT(*) FROM Tweets WHERE retweet = “no” ;

SELECT * FROM Tweets WHERE retweet IS NOT NULL ;
```

STORING A BITMAP INDEX

- One bitmap for each value, and one for NULL
- to store each bitmap, use one file for each
- Bitmaps can be compressed!

index size = #tuples * (domain size + 1) *bits*

When is a bitmap more space efficient than a B+ tree?

#distinct values < data entry size in the B+-tree

Bitslice Index

MOTIVATION

Reconsider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,       -- unique message id
    tstamp      TIMESTAMP,     -- when was the tweet posted
    uid         INTEGER,       -- unique id of the user
    msg         VARCHAR (140), -- the actual message
    zip          INTEGER,       -- zipcode when posted
    retweet      BOOLEAN       -- retweeted?
    );
```

with the following query:

```
SELECT * FROM Tweets WHERE zip = 53706 ;
```

Building a bitmap index on zip is not a good idea!

BITSLICE INDEX

table (stored in heapfile)

uniqueMsgID	…	zip	retweet
1	…	11324	yes
2	…	53705	yes
3	…	53706	no
4	…	53705	NULL
5	…	90210	no
…	…	…	…
1,000,000,000	…	53705	yes

1 slice per bit
+ (possibly) one more slice for NULL

bitslice index

00010110000111100
01101000111001001
01101000111001010
01101000111001001
10110000001100010
…
01101000111001001

slice 16
higher bit

slice 0
lower bit

BITSLICE INDEX: QUERIES

...	zip
...	11324
...	53705
...	53706
...	53705
...	90210
...	...
...	53705

00010110000111100
01101000111001001
01101000111001010
01101000111001001
10110000001100010
...
01101000111001001
00010111011100000

= **12000** in binary

slice 16 slice 0

```
SELECT * FROM Tweets
WHERE zip <= 12000 ;
```

1
0
0
0
0
...
0

walk through each slice constructing a **result bitmap**

- If we look for 0 and have 1, put 0 in the result
- If we look for 1 and have 0, put 1 in the result
- Else we need to consider the next bitslice

OTHER QUERIES

- We can also do aggregates with bitslice indices:
  - e.g. `SUM(attr)`: add bitslice by bitslice
    - count the number of 1s in slice 16 and multiply the count by 2^{16}
    - count the number of 1s in slice 15 and multiply the count by 2^{15}
    - …
- We can store each slice using methods like what we have for a bitmap (we can compress again!)

BITMAP VS BITSLICE INDEX

- Bitmaps are better for low cardinality domains
- Bitslices are better for high cardinality domains

- It is generally easier to “do the math” with bitmap indices

External Sorting

CS 564- Spring 2020

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

I/O aware algorithms for sorting

- External merge
  - a primitive for sorting
- External merge-sort
  - basic algorithm
  - optimizations

WHY SORTING?

- users often want the data sorted (**`ORDER BY`**)
- first step in bulk-loading a B+ tree
- used in duplicate elimination (why?)
- the sort-merge join algorithm (later in class) involves sorting as a first step

SORTING IN DATABASES

Why don't the standard sorting algorithms work for a database system?

- merge sort
- quick sort
- heap sort

The data typically does not fit in memory!

e.g. how do we sort 1TB of data with 8GB of RAM?

External Merge

EXTERNAL MERGE PROBLEM

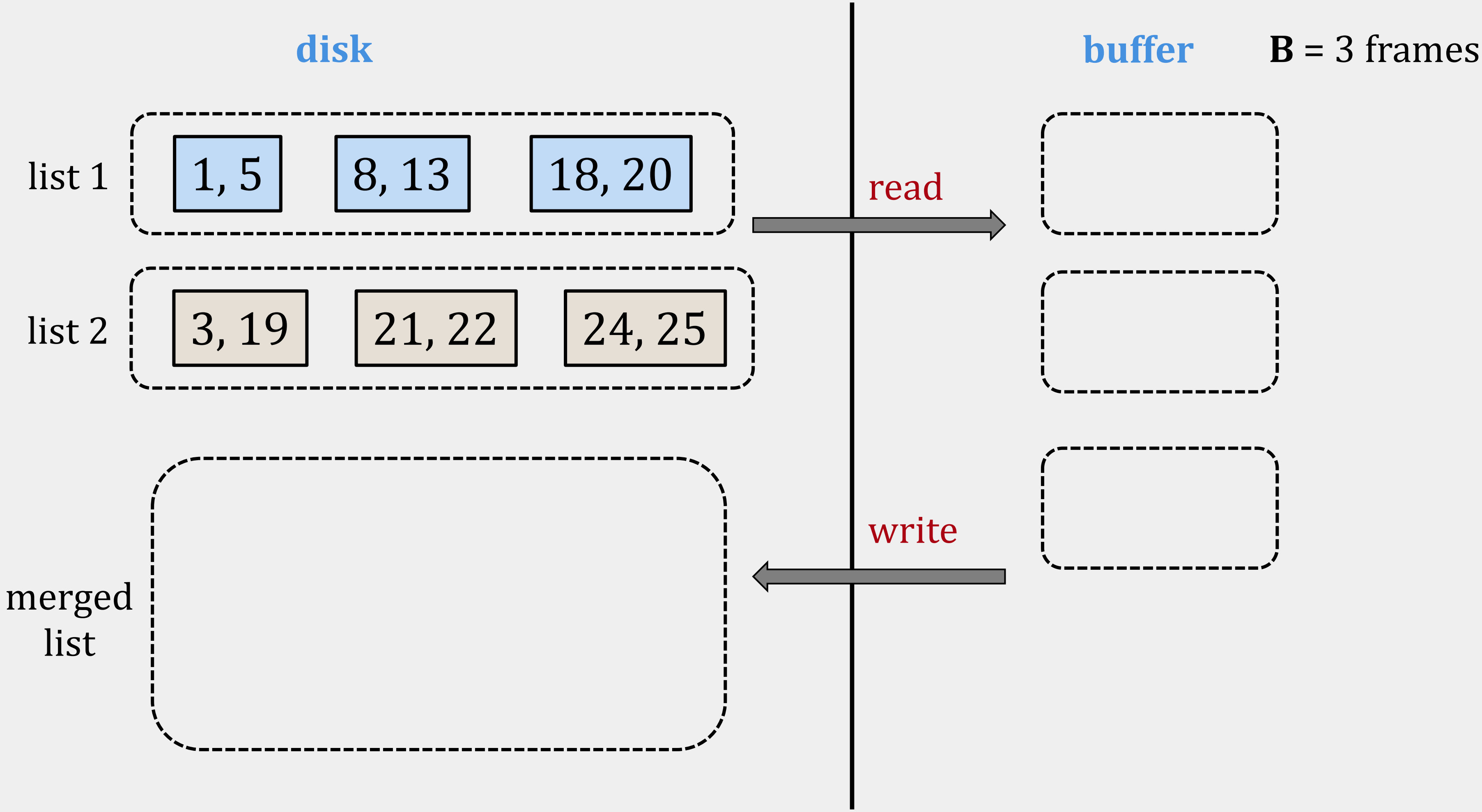
Input: 2 sorted lists (with M and N pages)

Output: 1 merged sorted list (with $M+N$ pages)

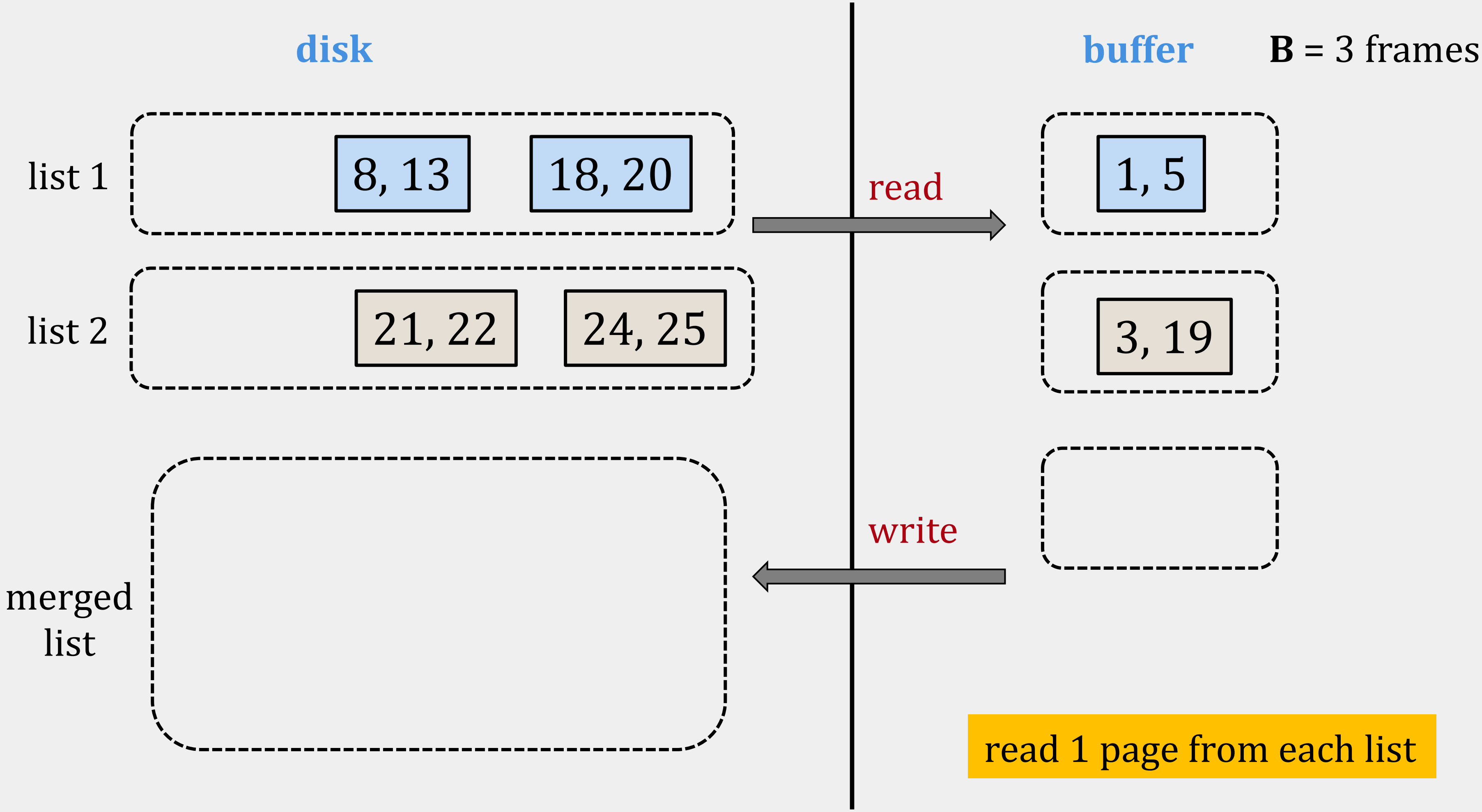
Can we efficiently (in terms of I/O) merge the two lists using a buffer of size at least *3*?

Yes, using only $2(M+N)$ I/Os !

EXTERNAL MERGE ALGORITHM

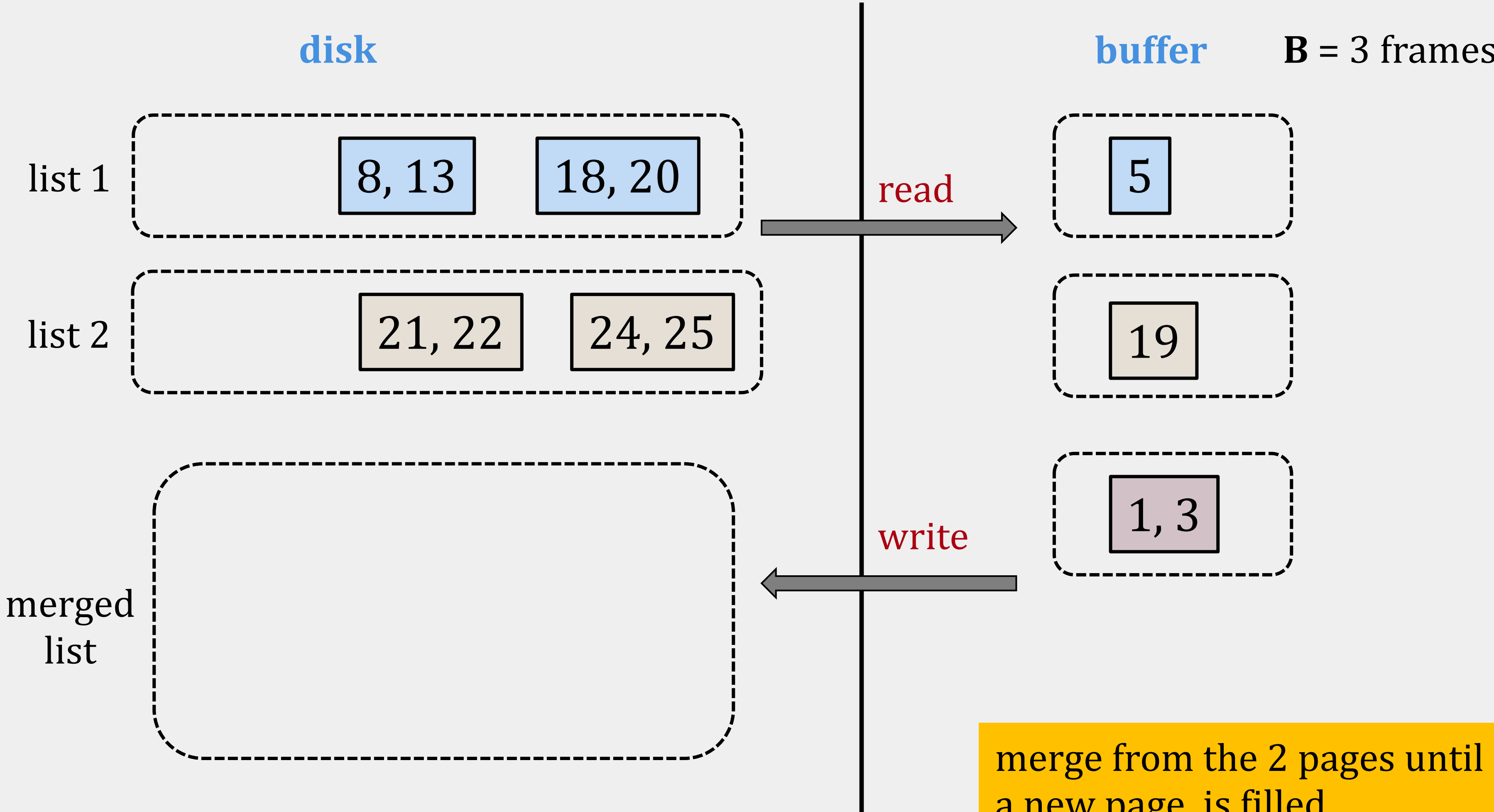


EXTERNAL MERGE ALGORITHM



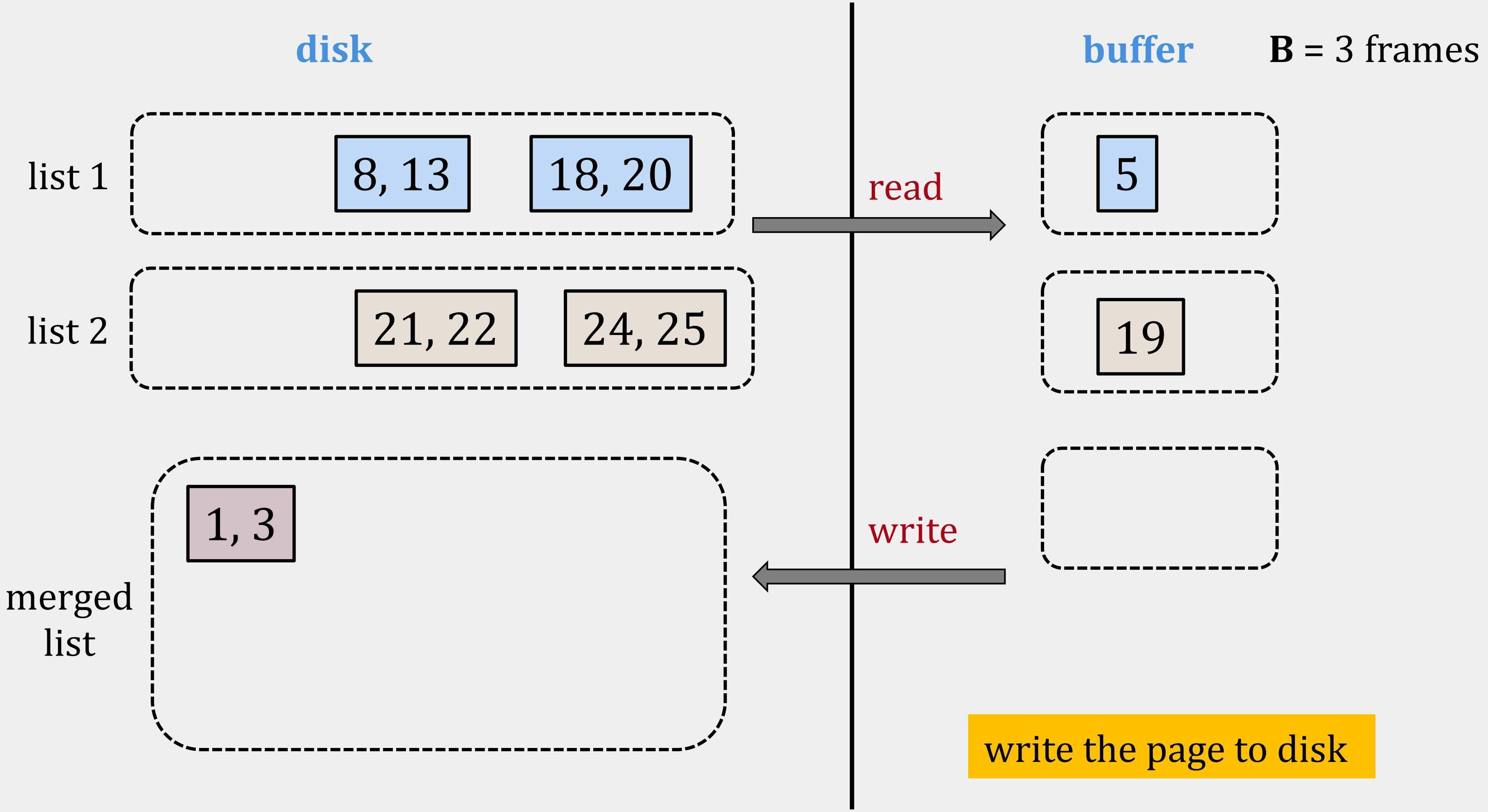
read 1 page from each list

EXTERNAL MERGE ALGORITHM

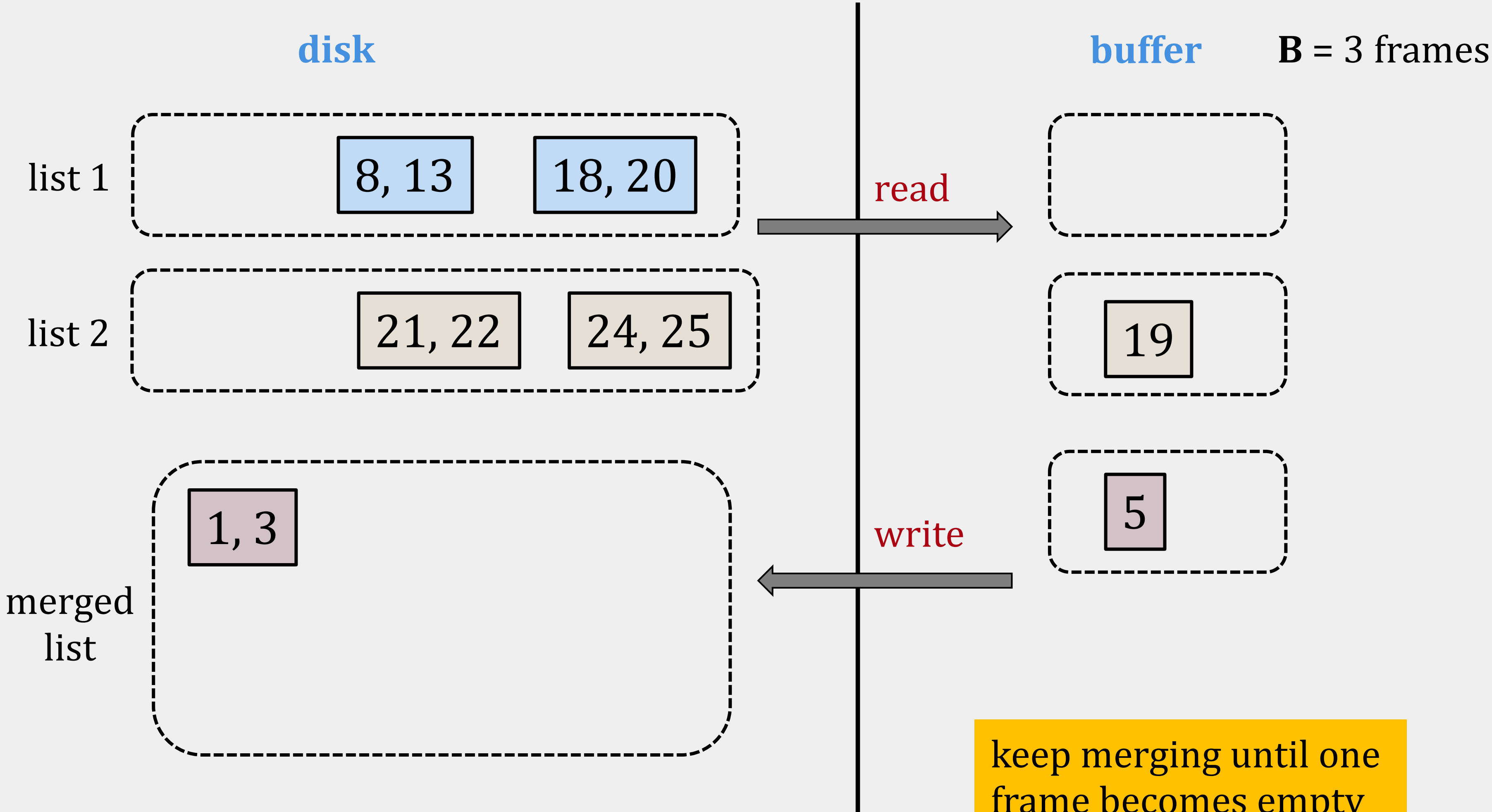


merge from the 2 pages until a new page is filled

EXTERNAL MERGE ALGORITHM

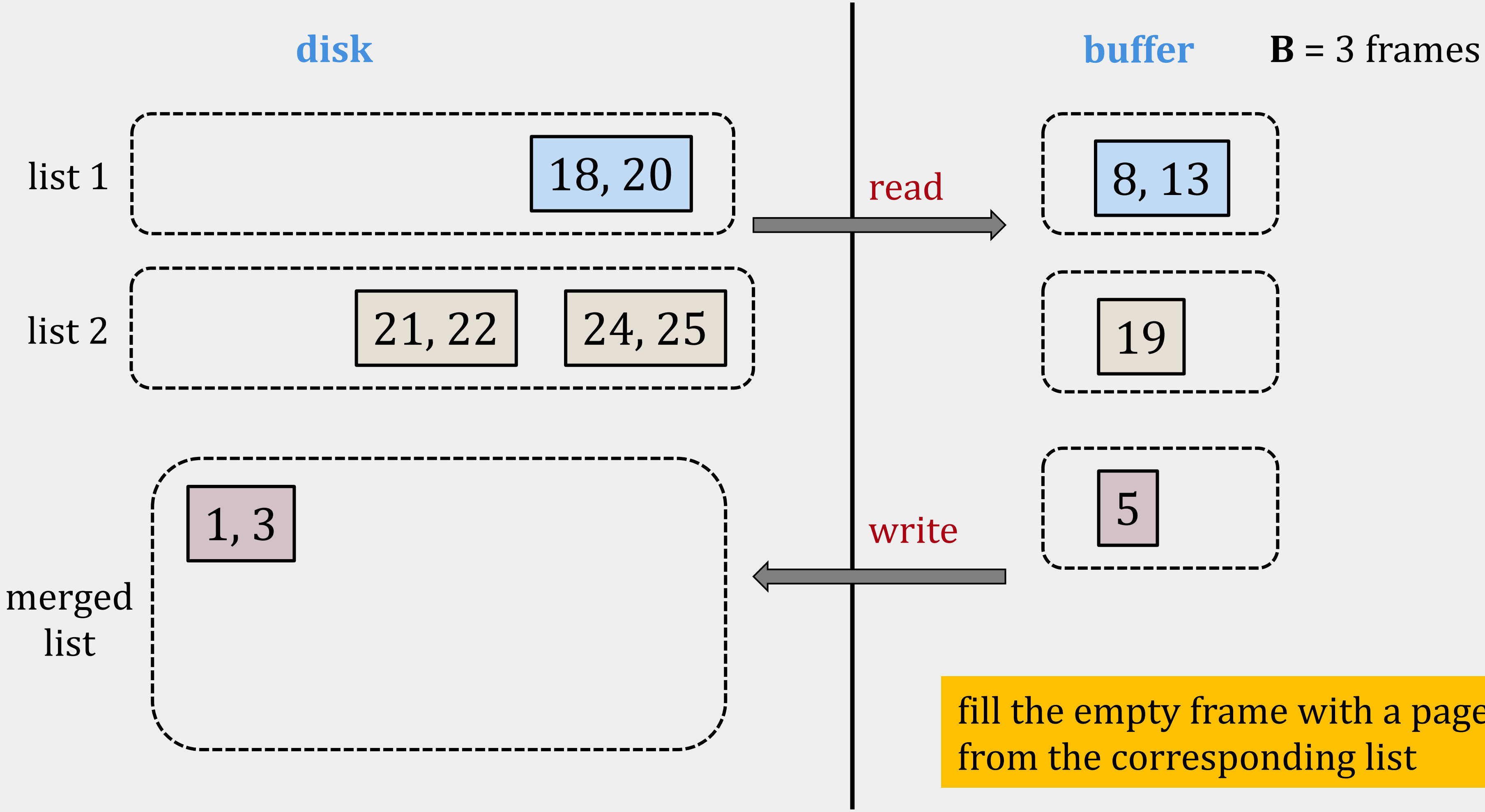


EXTERNAL MERGE ALGORITHM



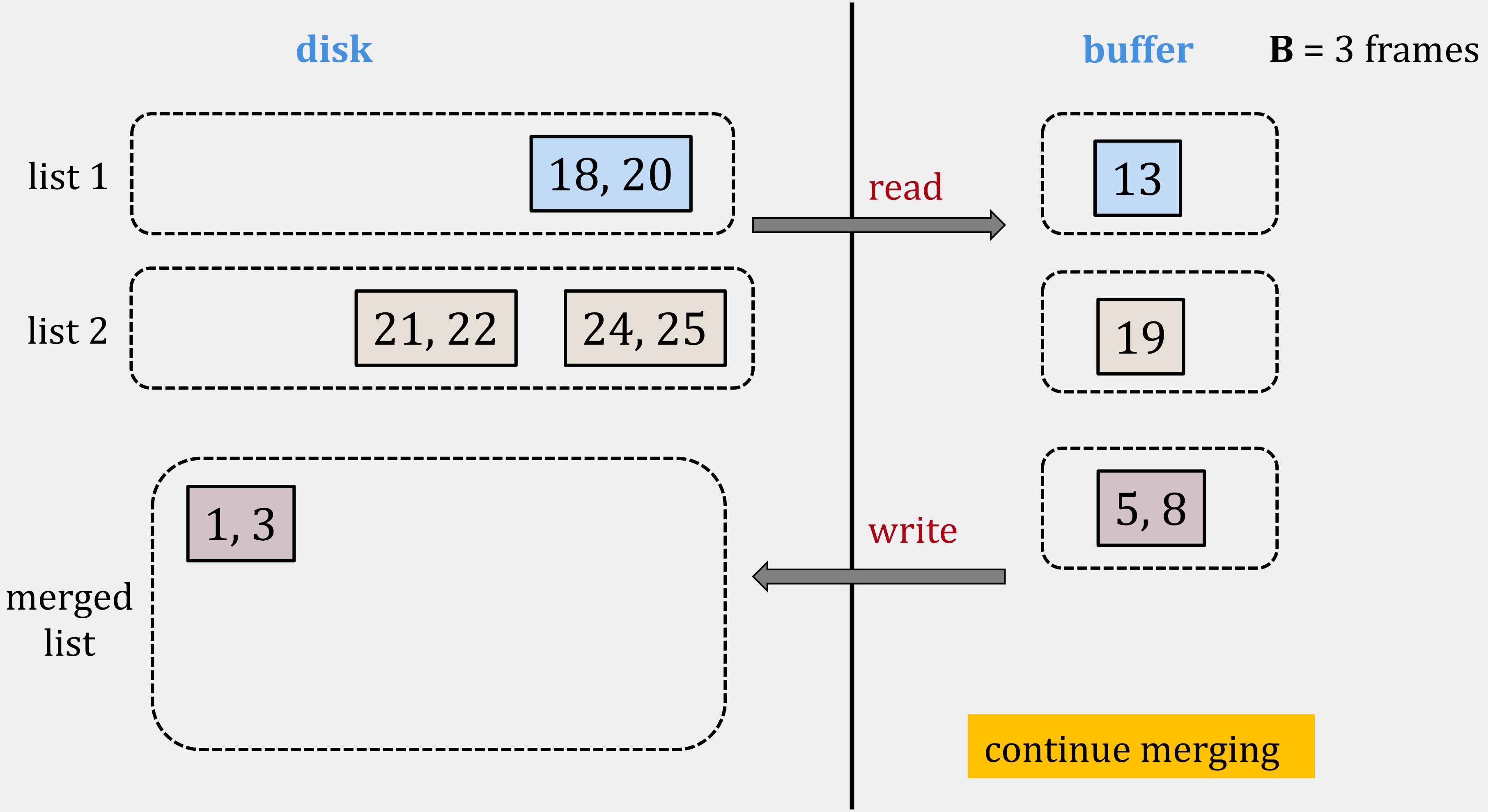
keep merging until one frame becomes empty

EXTERNAL MERGE ALGORITHM



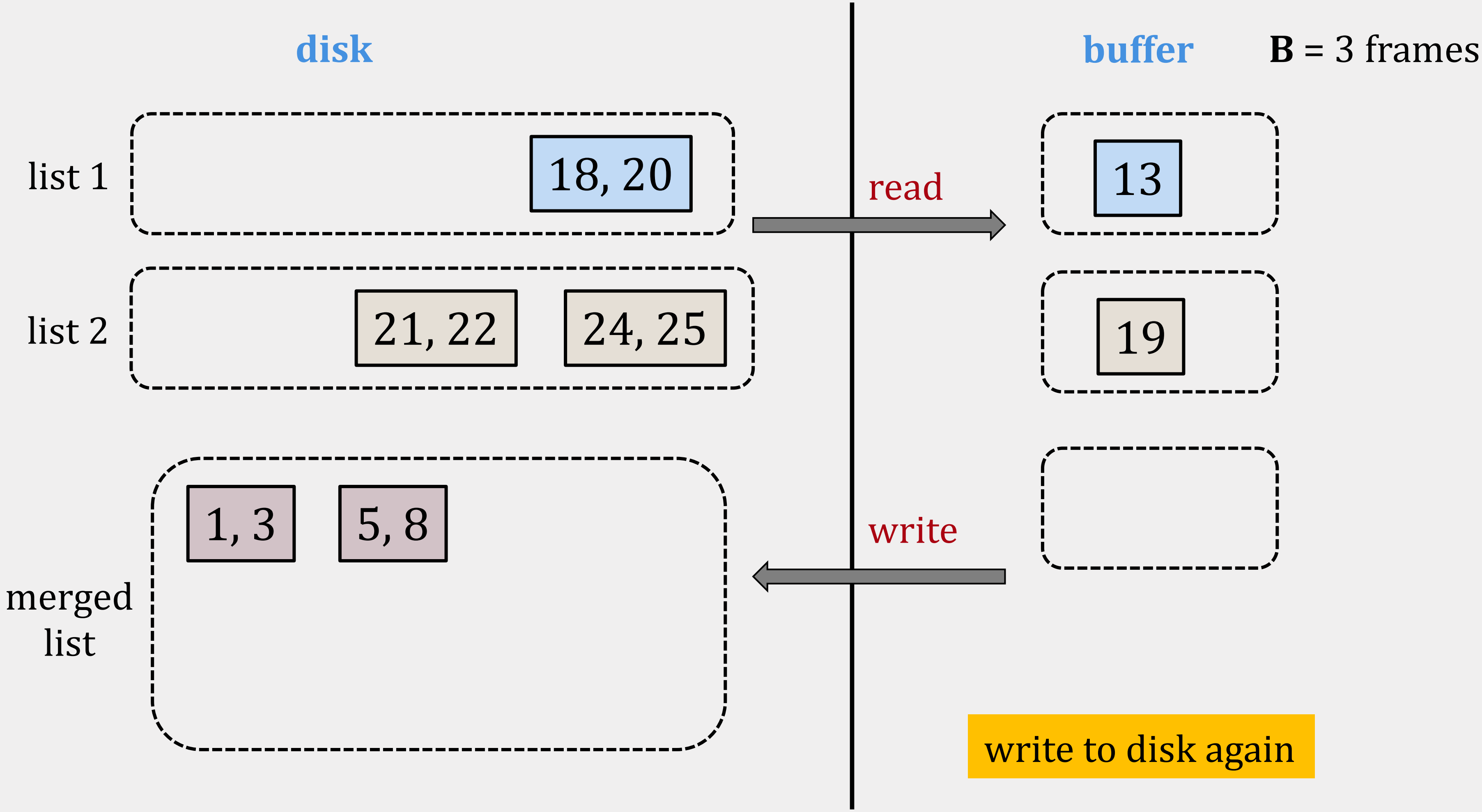
fill the empty frame with a page from the corresponding list

EXTERNAL MERGE ALGORITHM

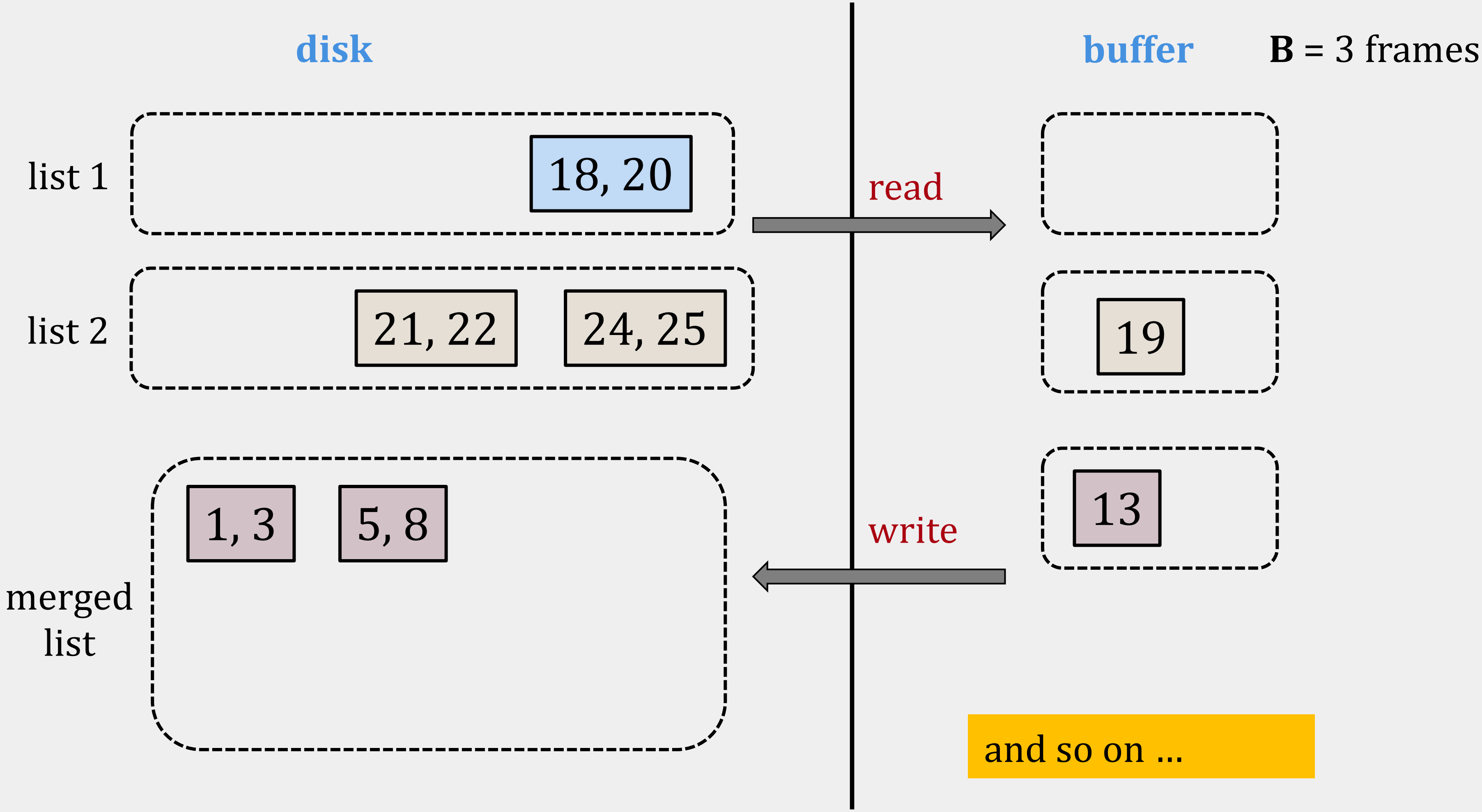


continue merging

EXTERNAL MERGE ALGORITHM



EXTERNAL MERGE ALGORITHM



EXTERNAL MERGE COST

We can merge 2 sorted lists of M and N pages using 3 buffer frames with

$$\text{I/O cost} = 2\ (M+N)$$

When we have $B+1$ buffer pages, we can merge B lists with the same I/O cost

External Merge Sort

THE SORTING PROBLEM

- **B** available pages in buffer pool
- a relation R of size **N** pages (where **N** > **B**)

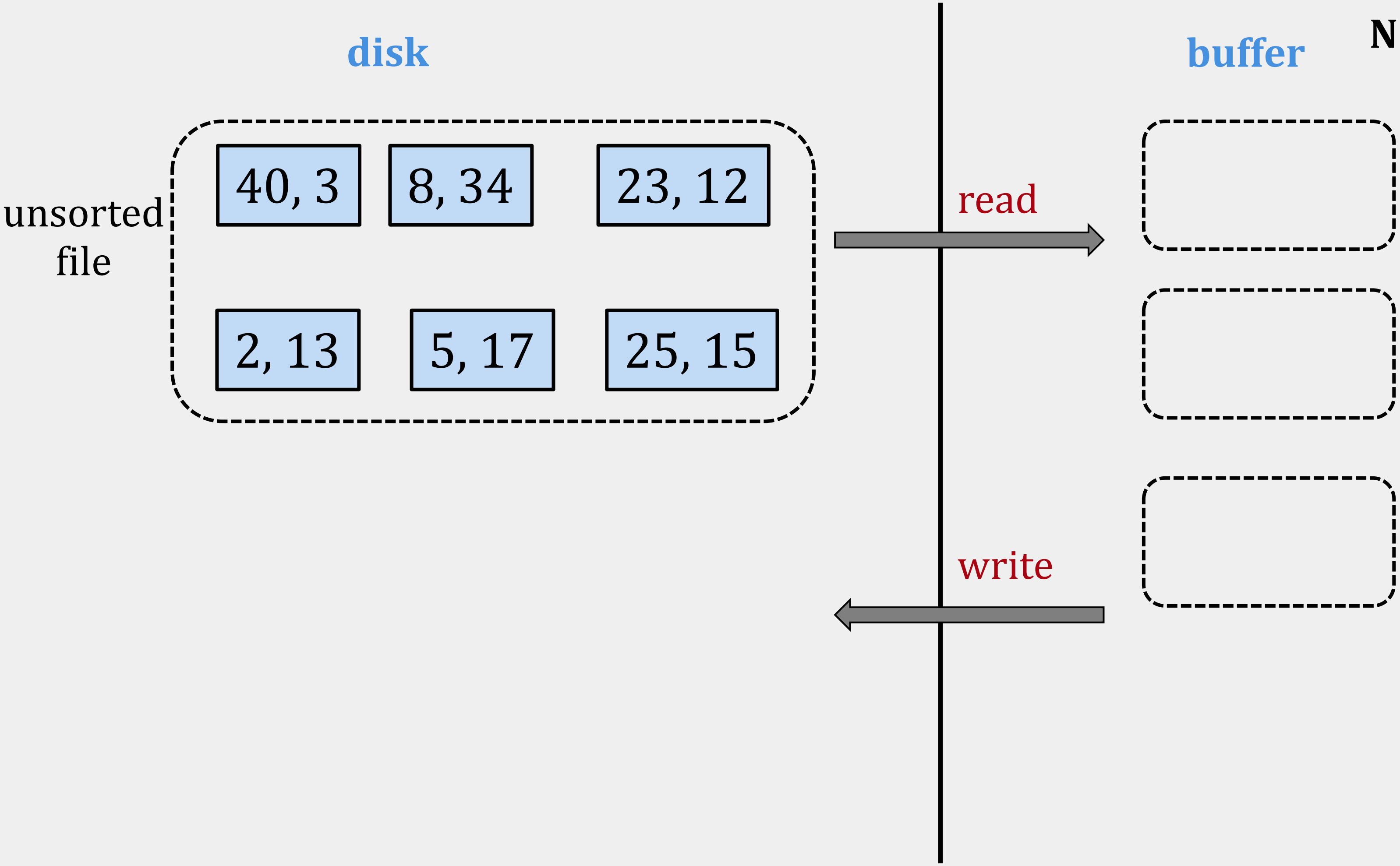
SORTING: output the same relation sorted on a given attribute

KEY IDEA

- split into chunks small enough to sort in memory (called runs)
- merge groups of runs using the external merge algorithm
- keep merging the resulting runs (each time is called a pass) until left with a single sorted file

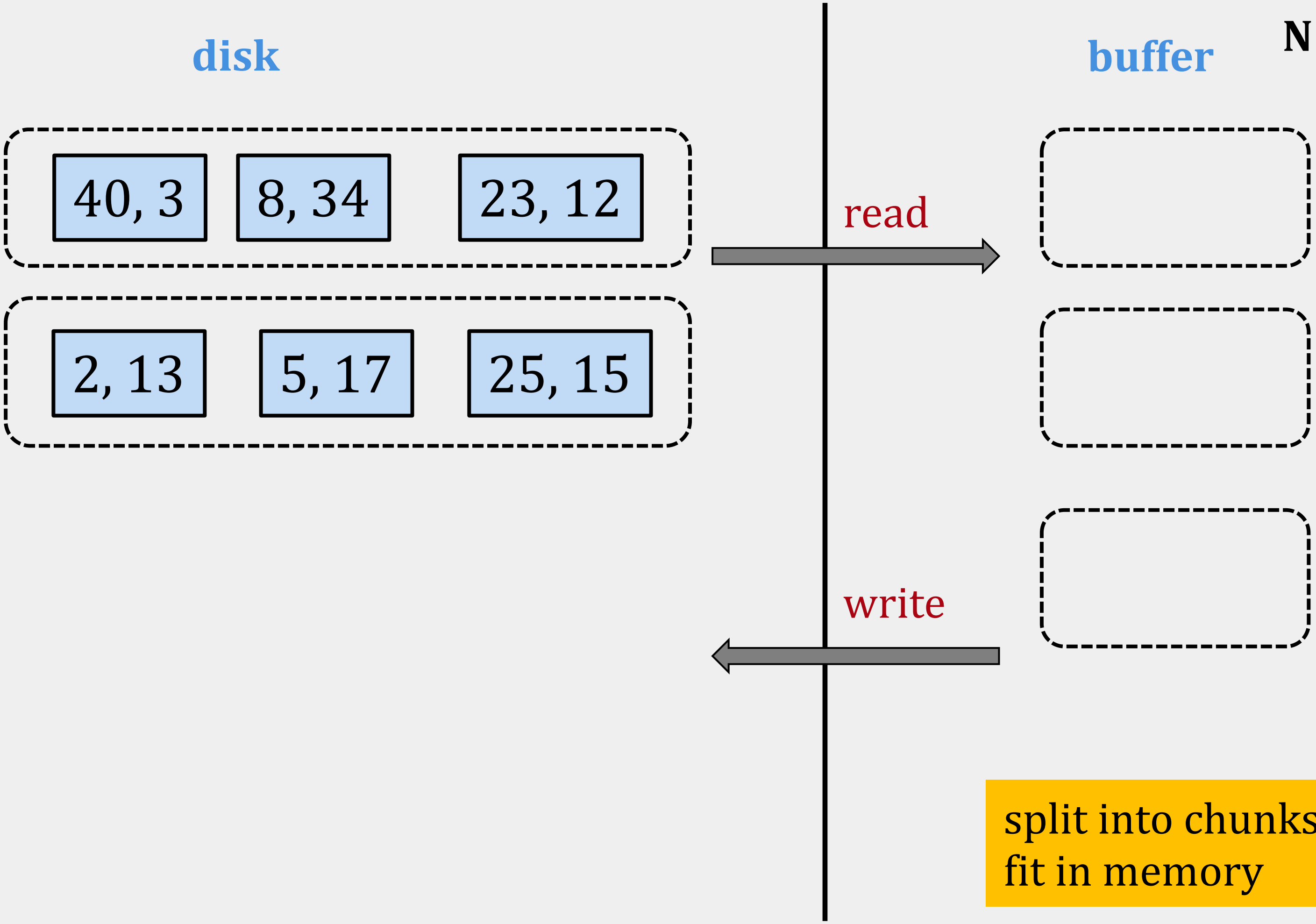
WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



WARM UP: 2-WAY SORT

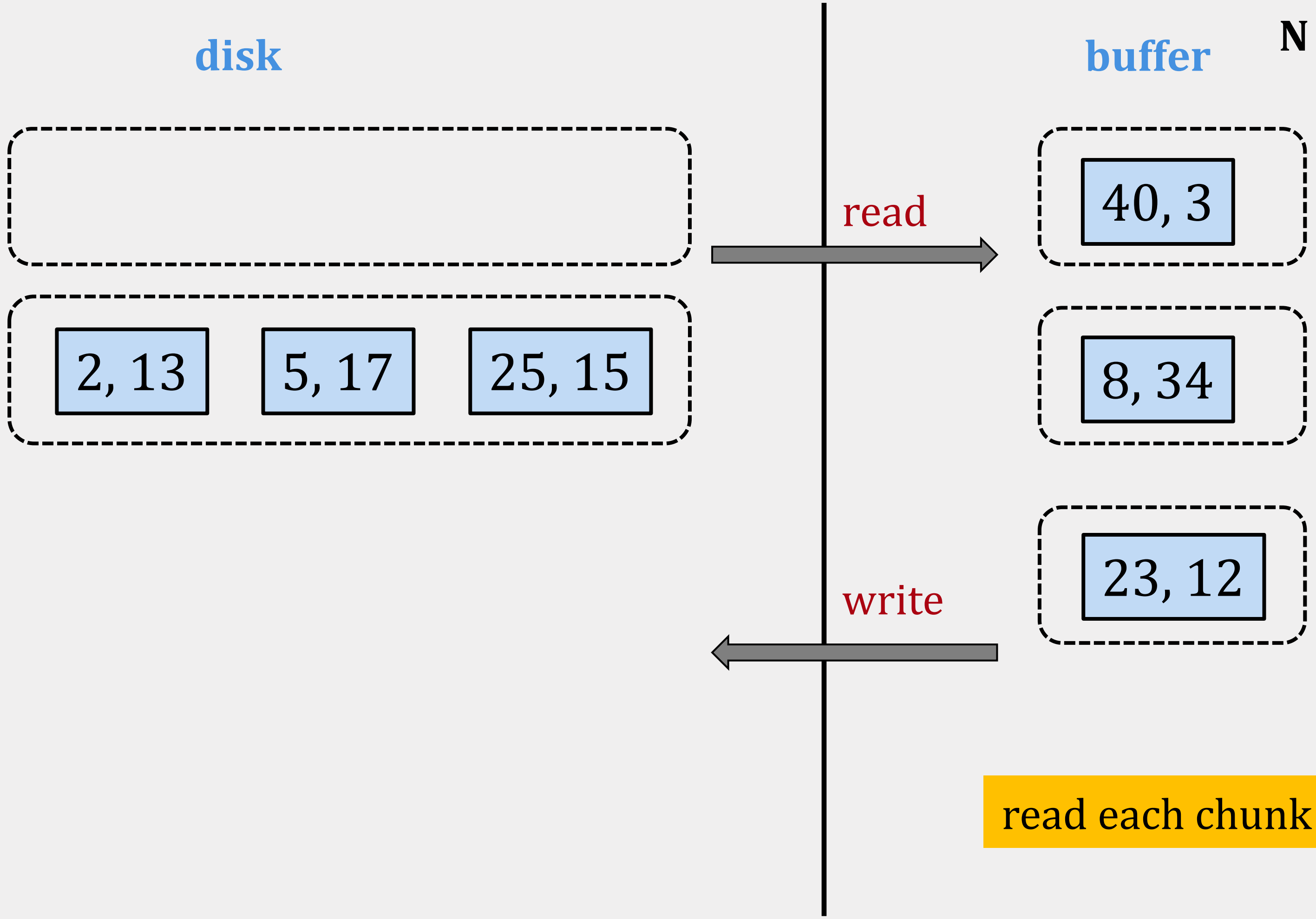
B = 3 frames
N = 6 pages



split into chunks that fit in memory

WARM UP: 2-WAY SORT

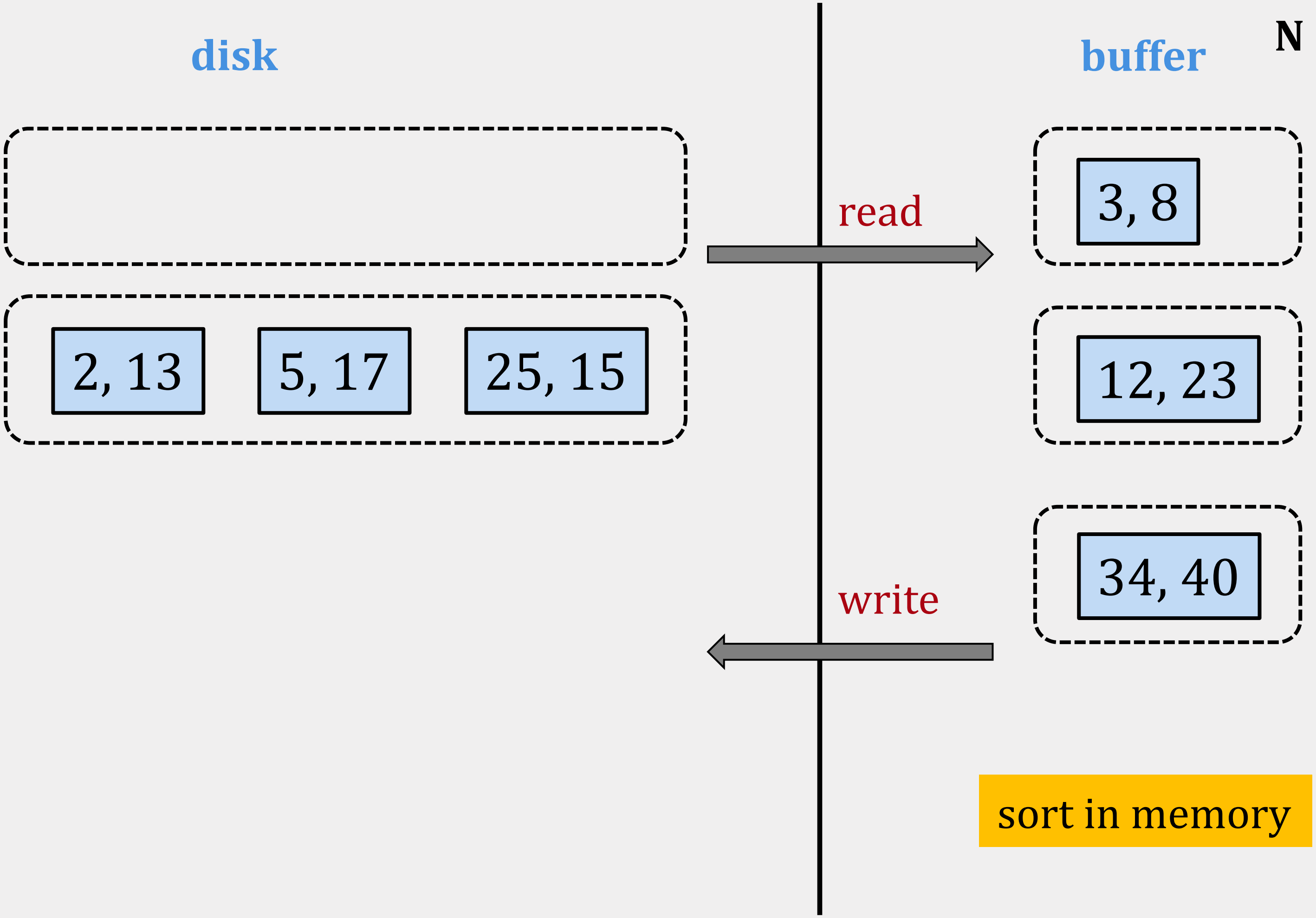
B = 3 frames
N = 6 pages



read each chunk in memory

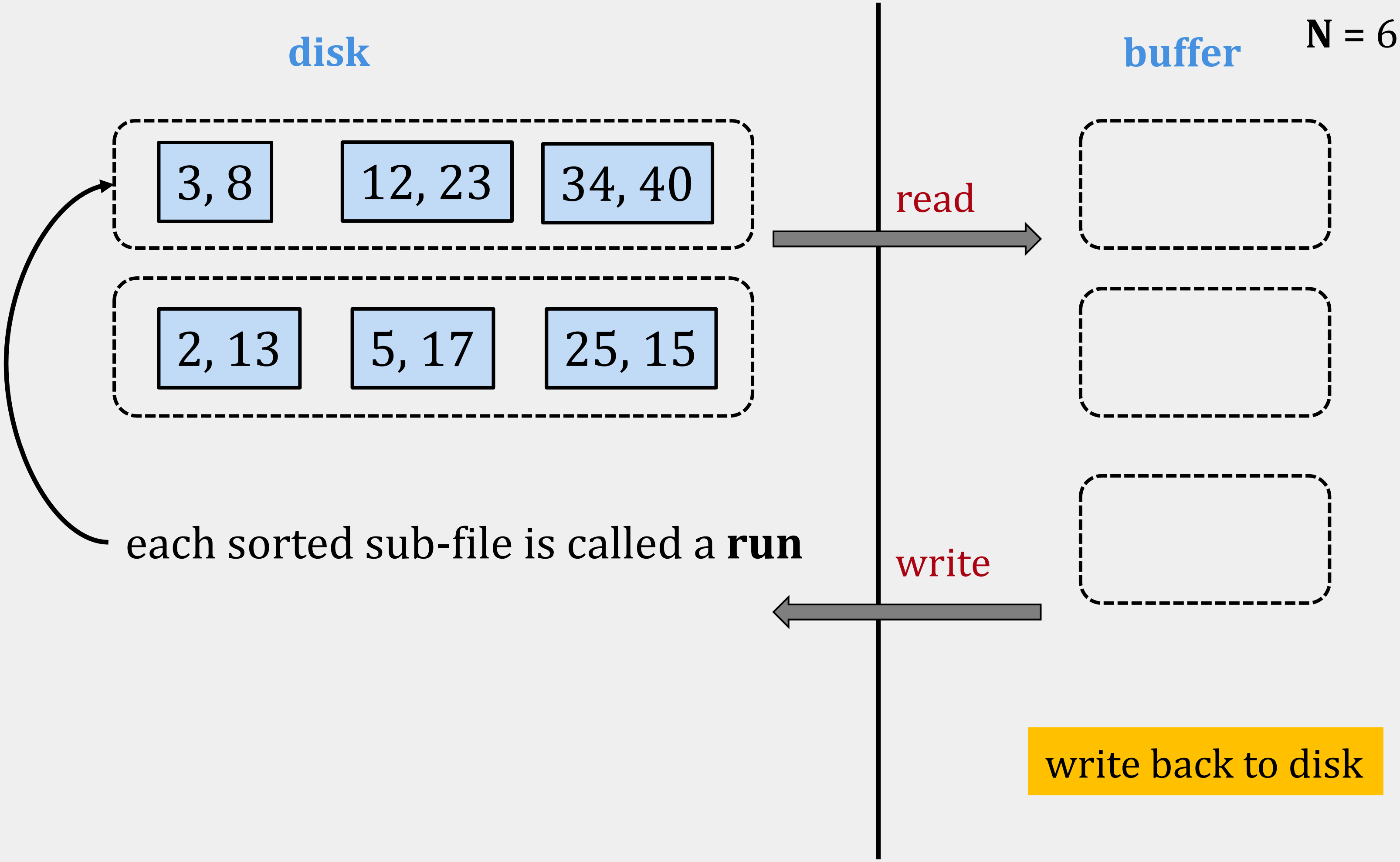
WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



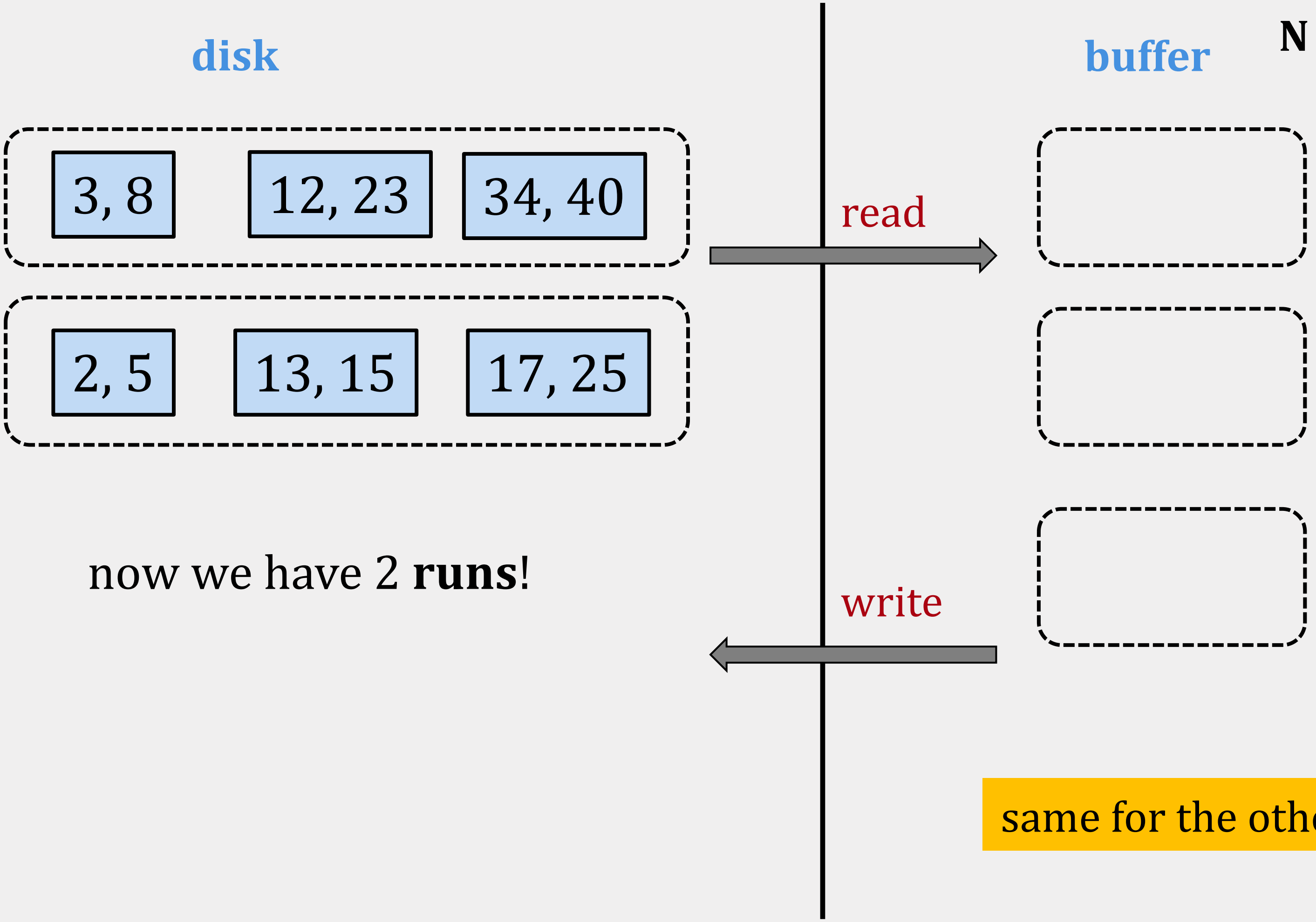
WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages

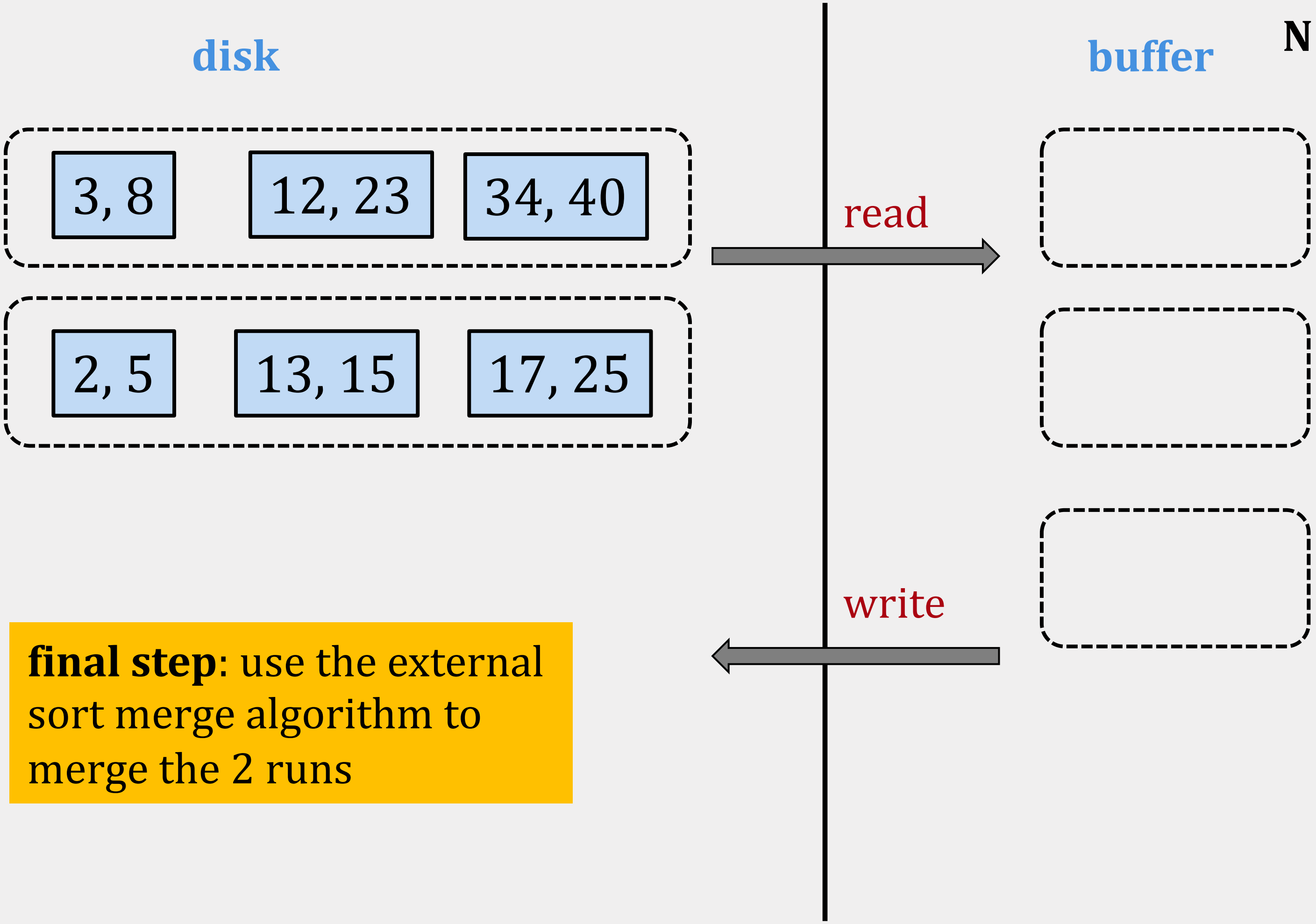


now we have 2 **runs**!

same for the other chunk

WARM UP: 2-WAY SORT

B = 3 frames
N = 6 pages



final step: use the external sort merge algorithm to merge the 2 runs

CALCULATING THE I/O COST

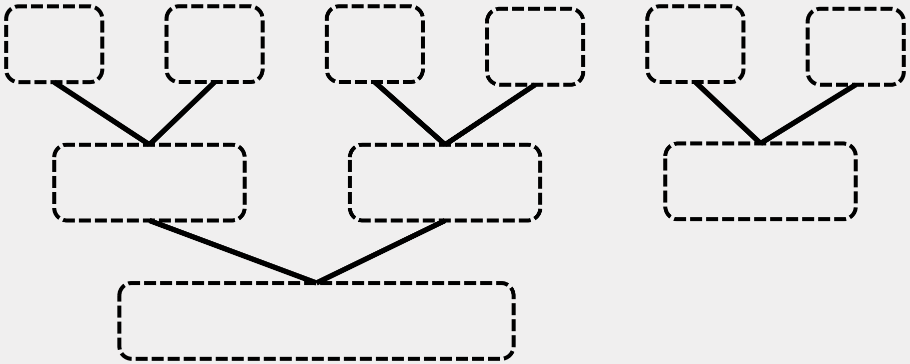
In our example, **B**= 3 buffer pages, **N** = 6 pages

- Pass **0**: creating the first runs
  - 1 read + 1 write for every page
  - total cost = $6 * (1 + 1) = 12$ I/Os
- Pass **1**: external merge sort
  - total cost = $2 * (3 + 3) = 12$ I/Os

So 24 I/Os in total

I/O COST: SIMPLIFIED VERSION

Assume for now that we initially create **N** runs, each run consisting of a single page



pass 0: N runs, each 1 page

pass 1: merge into N/2 runs

pass 2: merge into N/4 runs

- We need $\lceil log_2 N \rceil + 1$ passes to sort the whole file
- Each pass needs $2N$ I/Os

total I/O cost = $2N(\lceil log_2 N \rceil + 1)$

CAN WE DO BETTER?

- The 2-way merge algorithm only uses 3 buffer pages
- But we have more available memory!

Key idea: use as much of the available memory as possible in every pass

- reducing the number of passes reduces I/O

EXTERNAL SORT: I/O COST

Suppose we have $B \geq 3$ buffer pages available

$$2N(\lceil log_2 N \rceil + 1) \Longrightarrow 2N(\left\lceil log_2 \frac{N}{B} \right\rceil + 1) \Longrightarrow 2N(\left\lceil log_{B-1} \frac{N}{B} \right\rceil + 1)$$

- initial runs of length 1
- 3-way merge

increase the length of the initial runs to B

merge B-1 runs at a time

NUMBER OF PASSES

N	B=3	B=17	B=257
100	7	2	1
10,000	13	4	2
1,000,000	20	5	3
10,000,000	23	6	3
100,000,000	26	7	4
1,000,000,000	30	8	4

Optimizing Merge Sort

REPLACEMENT SORT

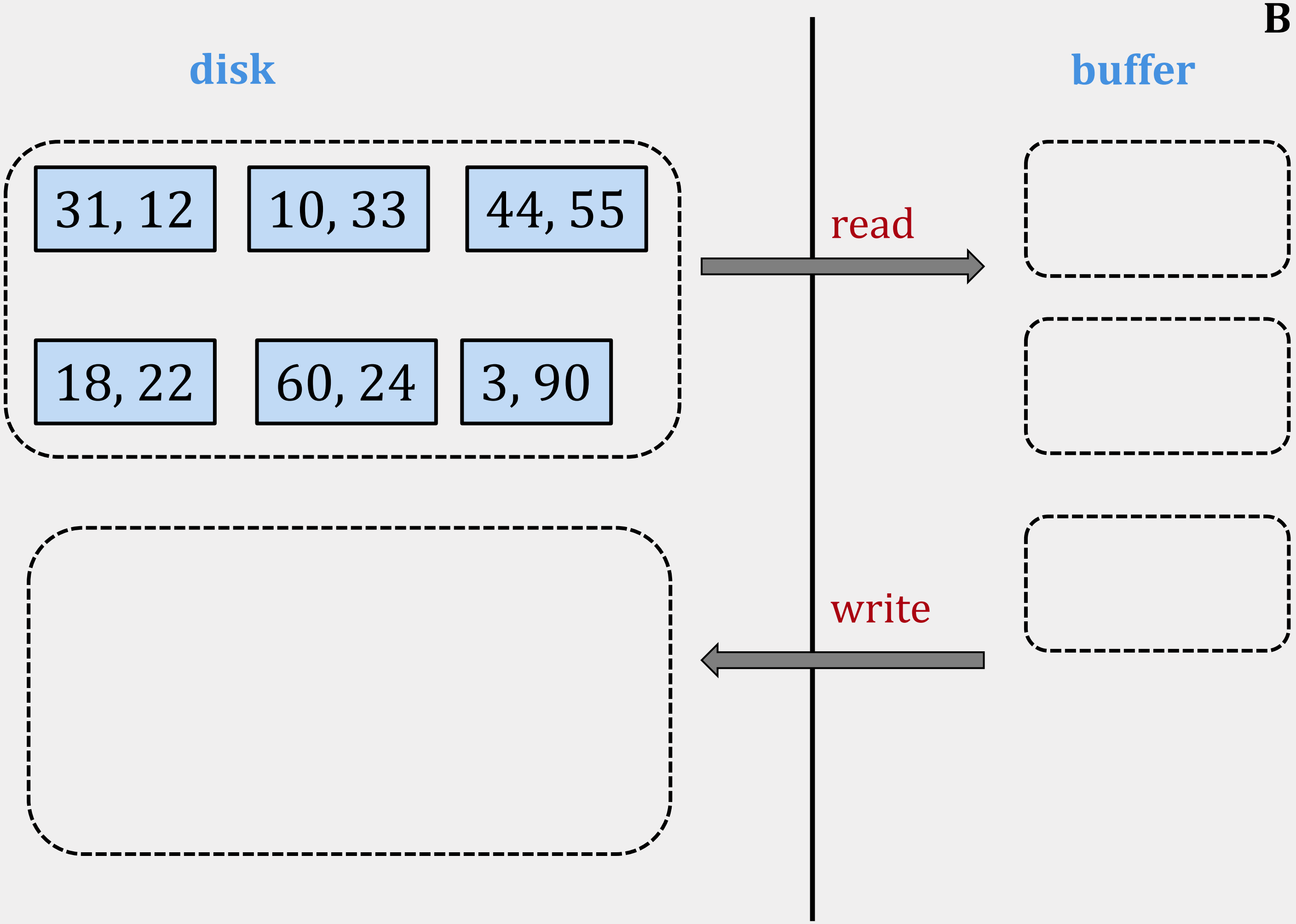
- used as an alternative for the sorting in pass 0
- creates runs of *average* size $2B$ (instead of B)

Algorithm

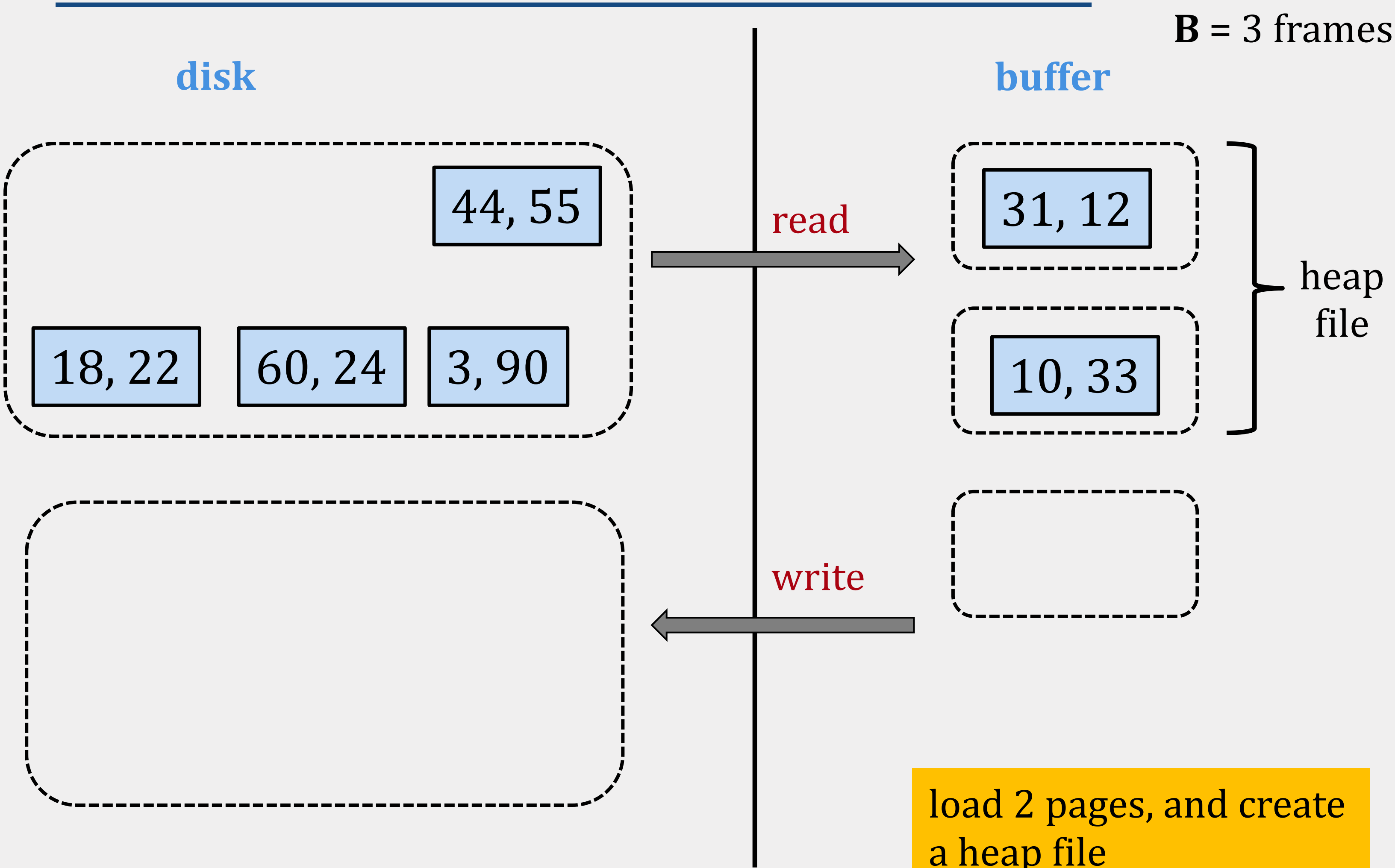
- read *B-1* pages in memory (keep as sorted heap)
- move smallest record (that is greater than the largest element in buffer) to output buffer
- read a new record *r* and insert into the sorted heap

REPLACEMENT SORT: EXAMPLE

B = 3 frames

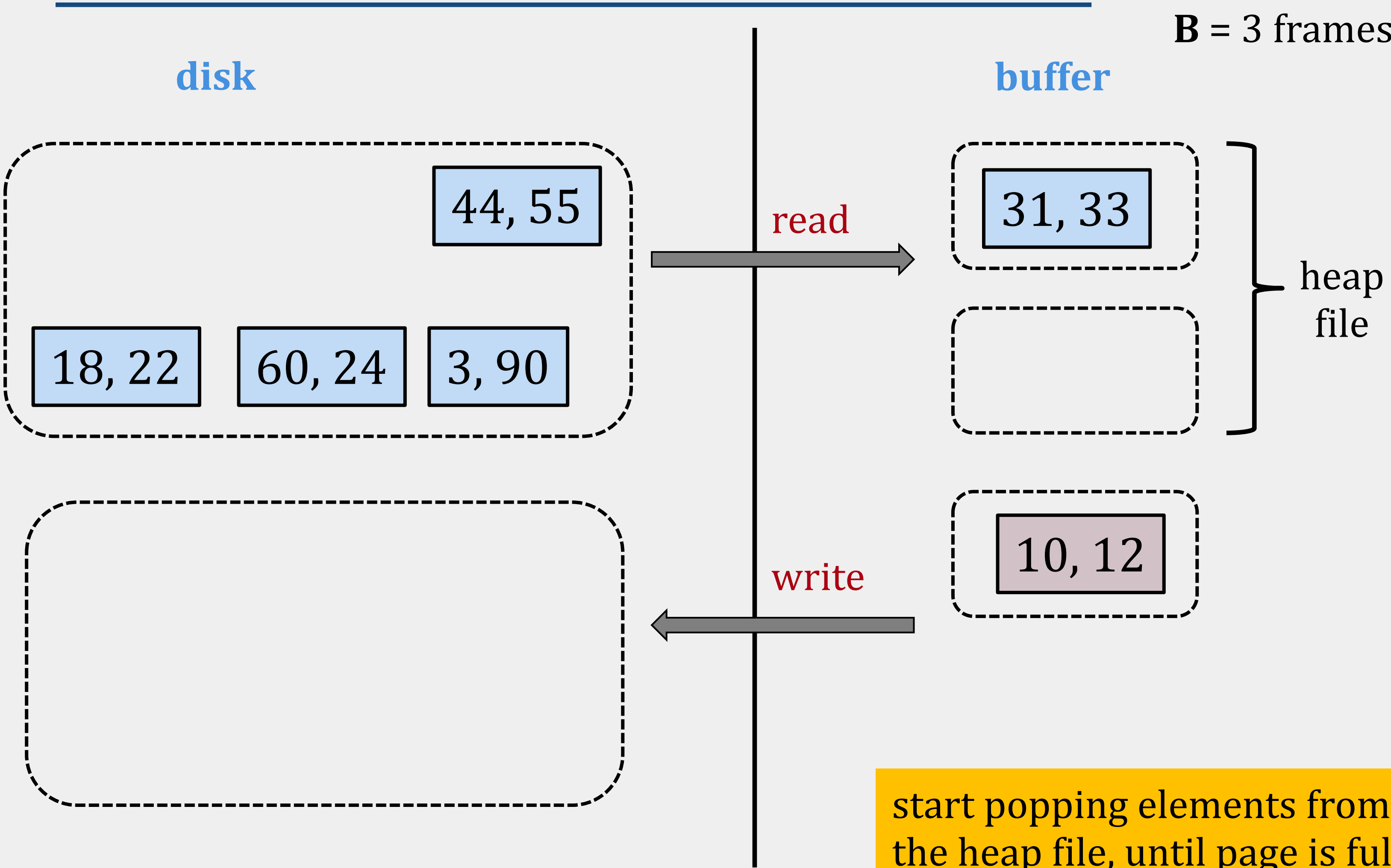


REPLACEMENT SORT: EXAMPLE



load 2 pages, and create a heap file

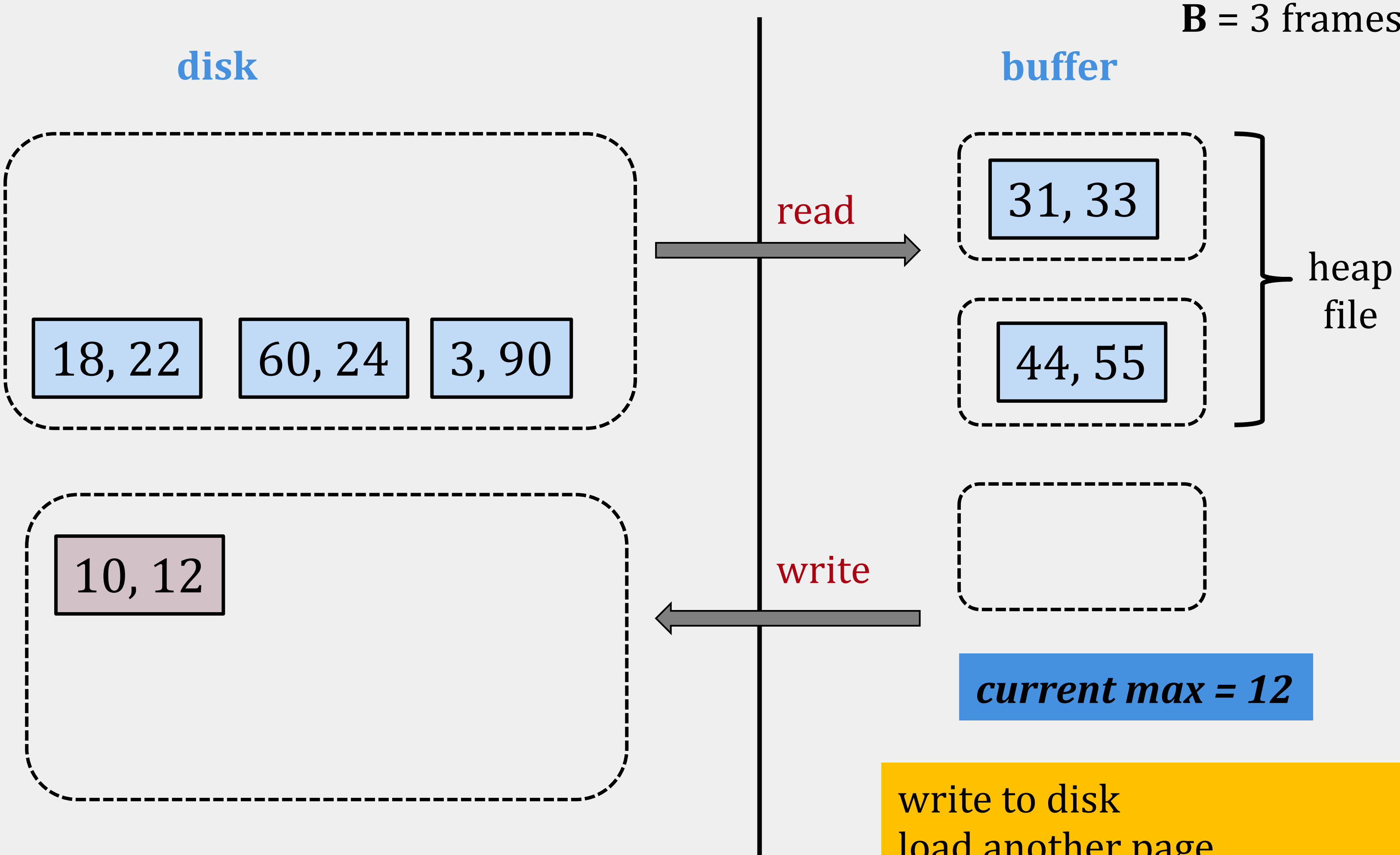
REPLACEMENT SORT: EXAMPLE



start popping elements from the heap file, until page is full

REPLACEMENT SORT: EXAMPLE

B = 3 frames

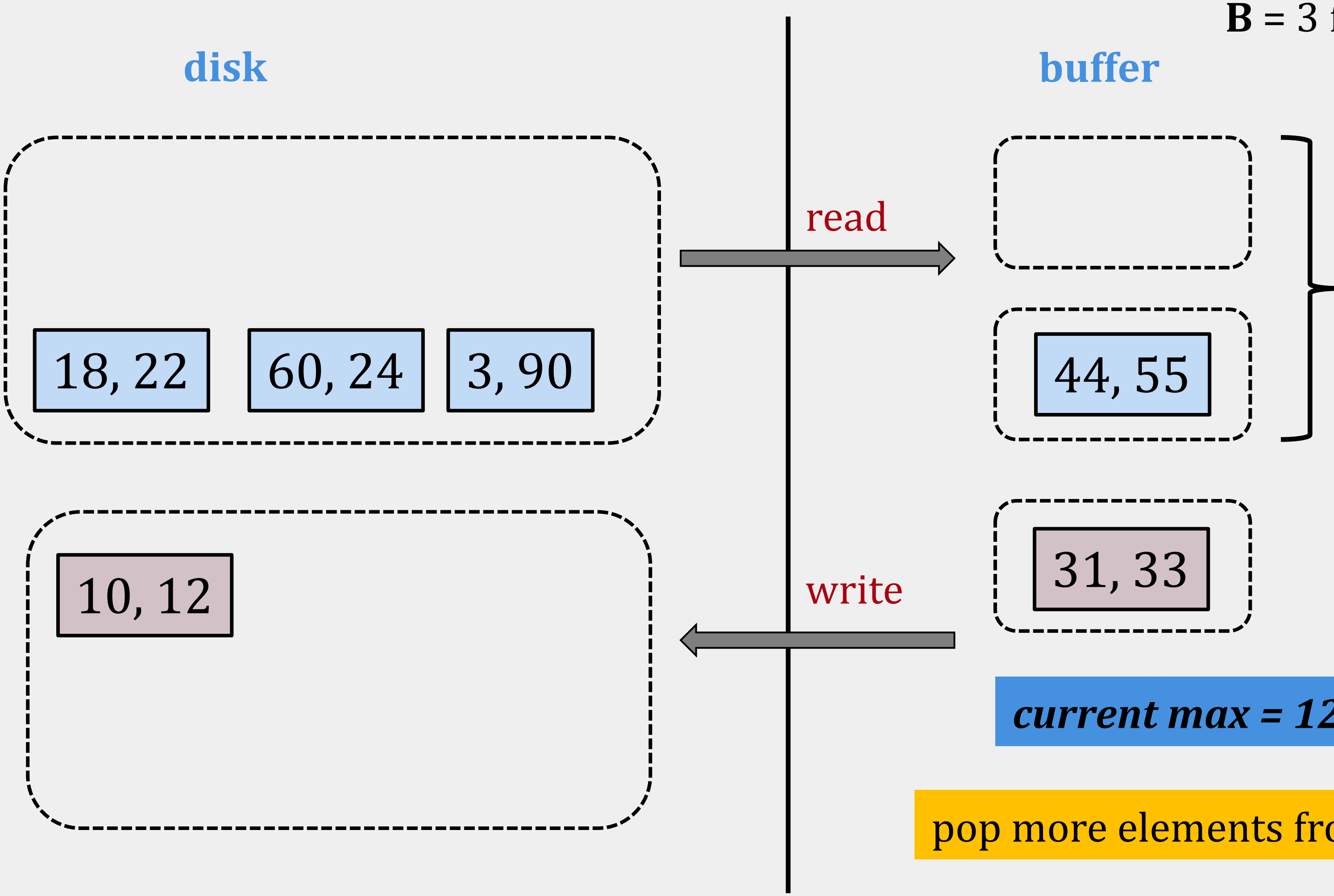


current max = 12

write to disk
load another page

REPLACEMENT SORT: EXAMPLE

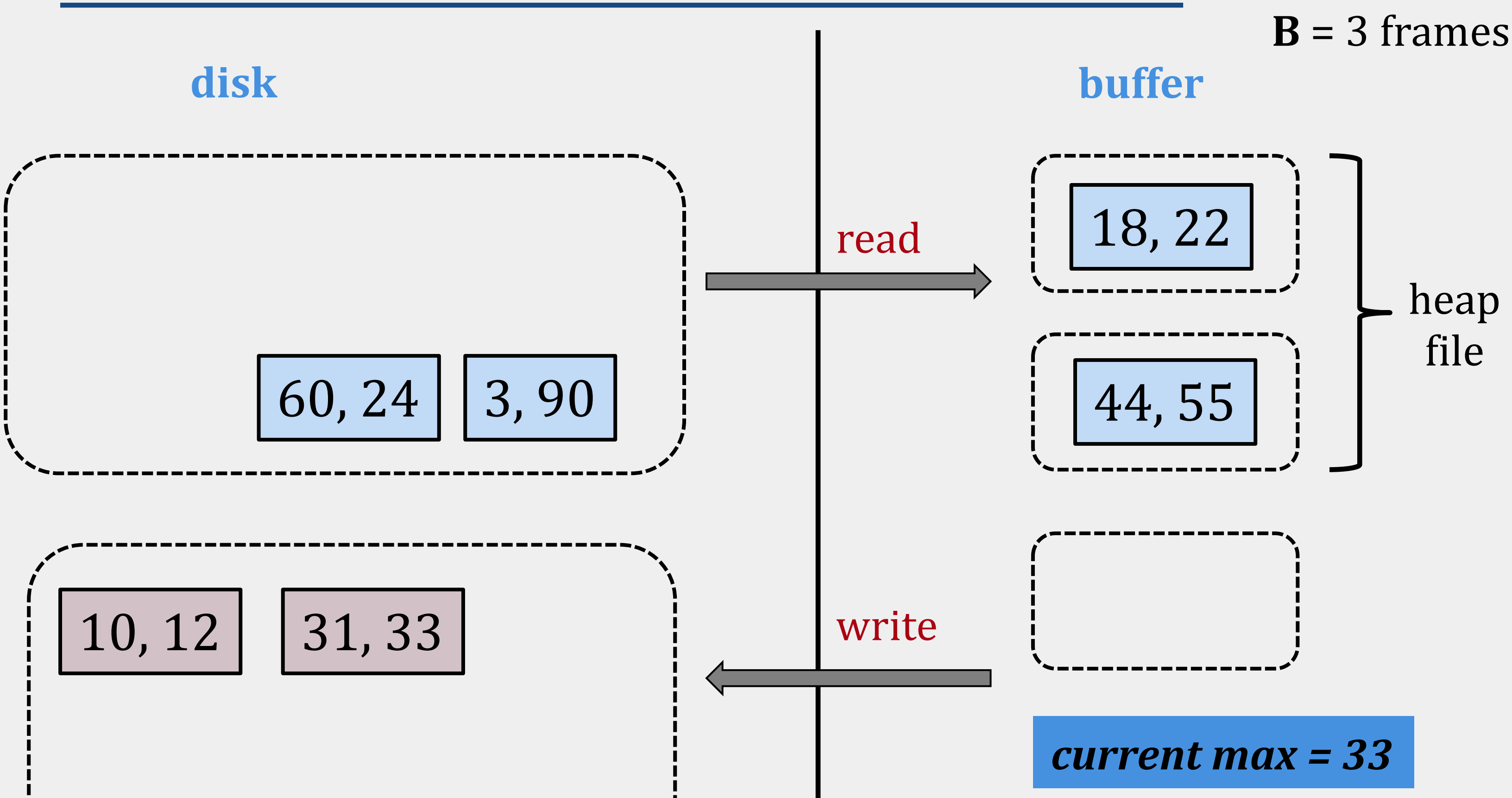
B = 3 frames



current max = 12

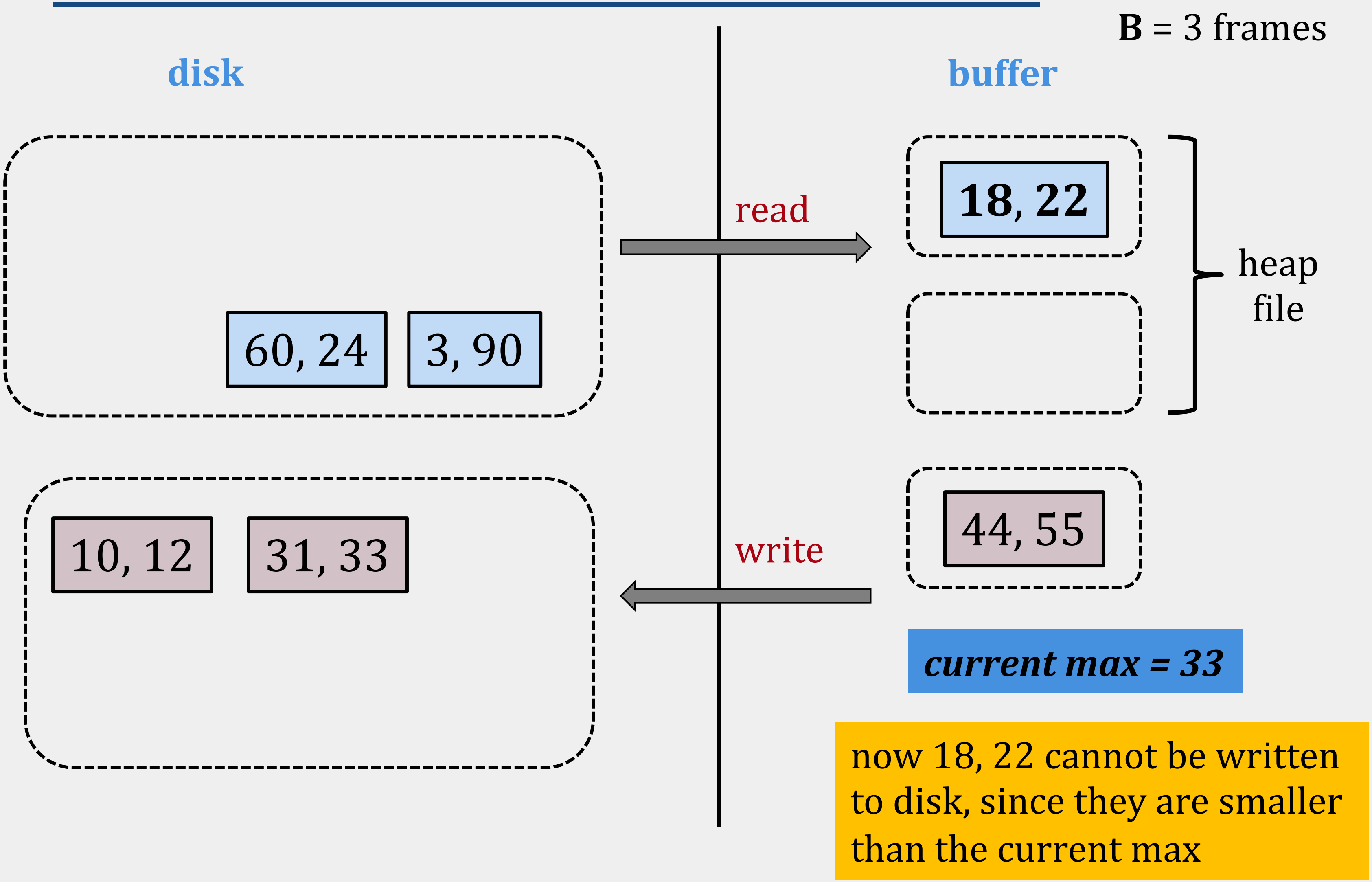
pop more elements from heap

REPLACEMENT SORT: EXAMPLE



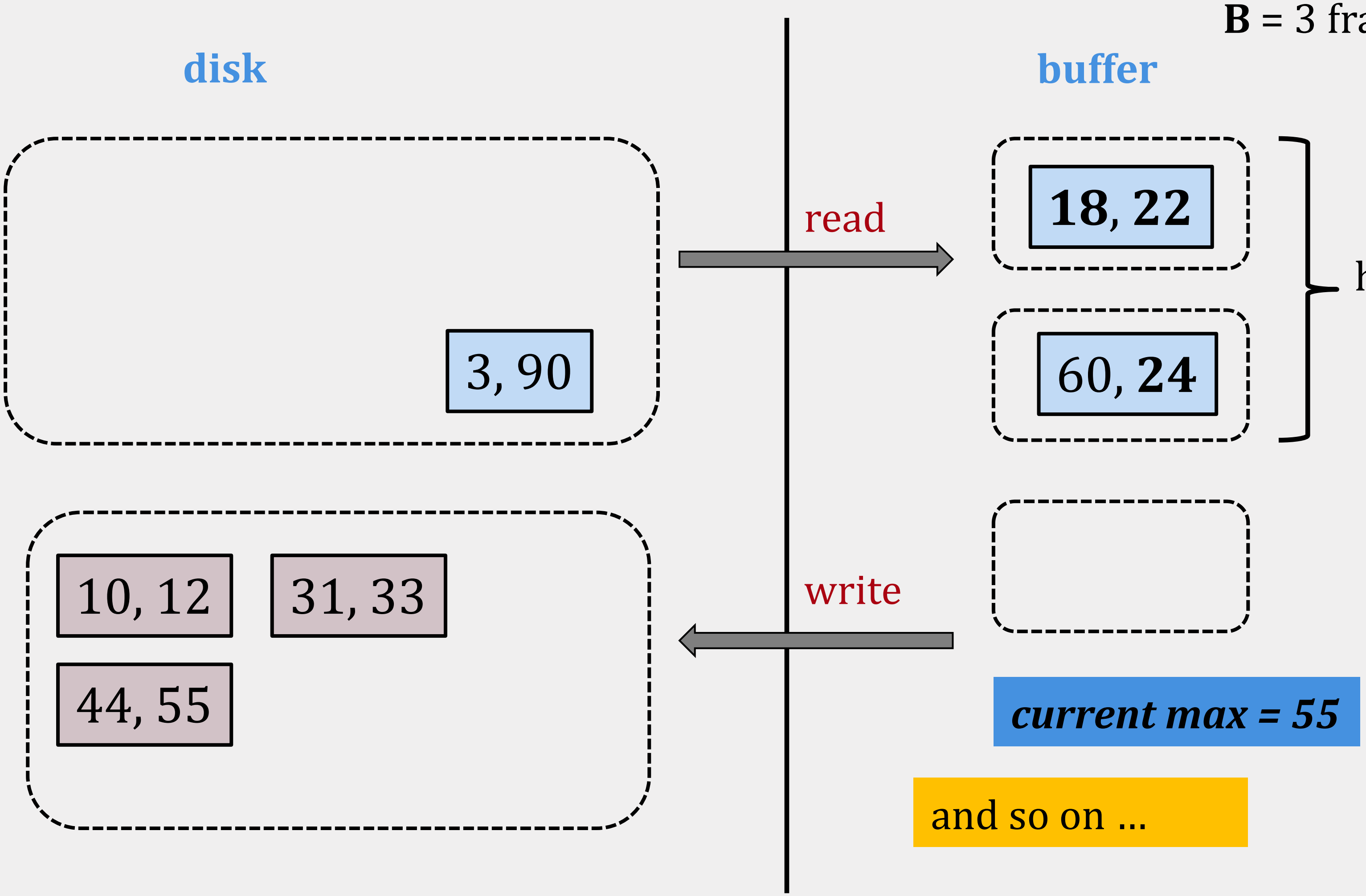
write to disk
load one more page

REPLACEMENT SORT: EXAMPLE

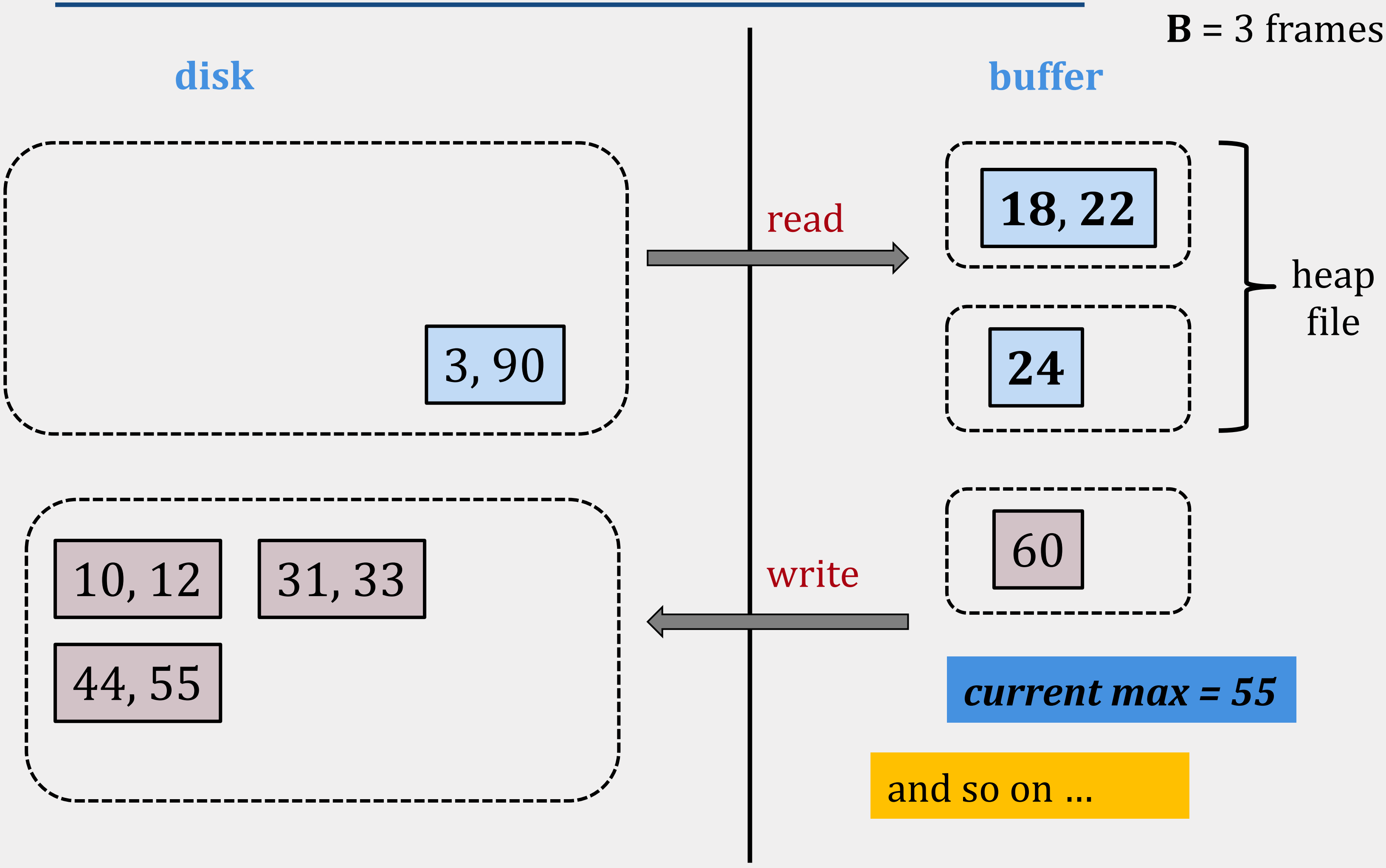


REPLACEMENT SORT: EXAMPLE

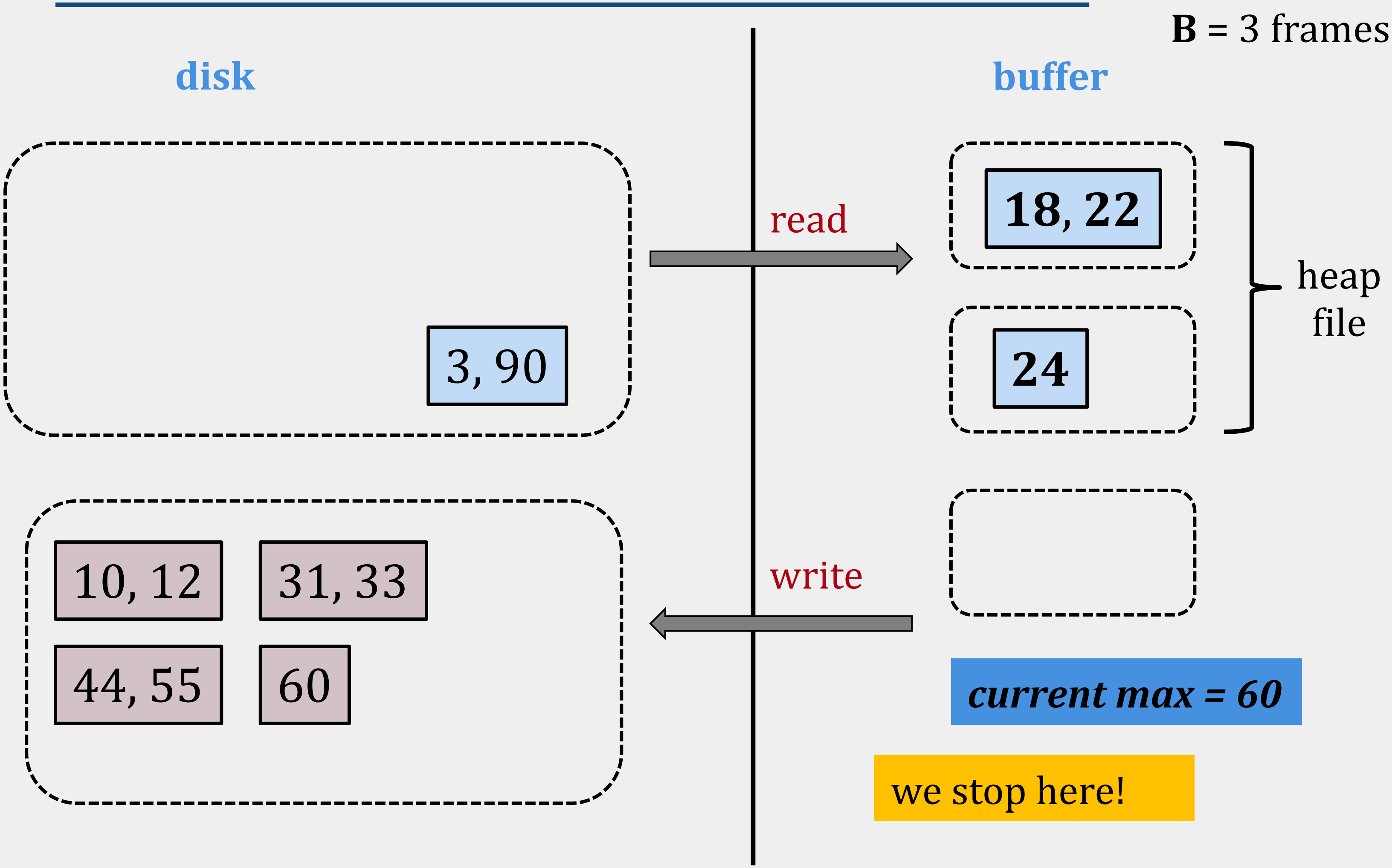
B = 3 frames



REPLACEMENT SORT: EXAMPLE



REPLACEMENT SORT: EXAMPLE



I/O COST WITH REPLACEMENT SORT

Each initial run has length $\sim 2B$

$$\text{I/O cost} = 2N\left(\left\lceil log_{B-1}\frac{N}{2B}\right\rceil + 1\right)$$

Relational Operators #1

CS 564- Spring 2020

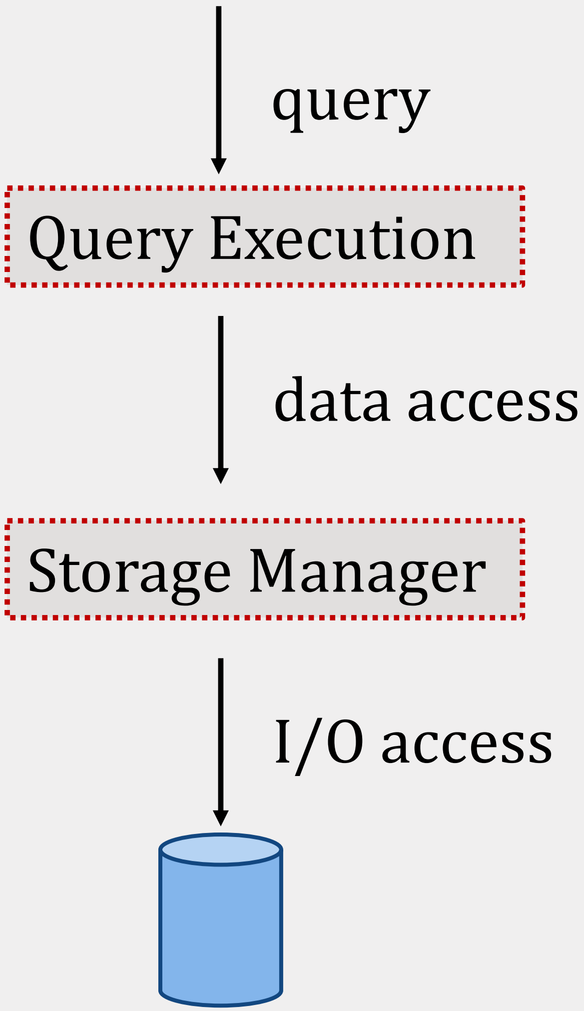
ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

Algorithms for relational operators:

- select
- project

ARCHITECTURE OF A DBMS



LOGICAL VS PHYSICAL OPERATORS

- Logical operators
  - *what* they do
  - e.g., union, selection, project, join, grouping

- Physical operators
  - *how* they do it
  - e.g., nested loop join, sort-merge join, hash join, index join

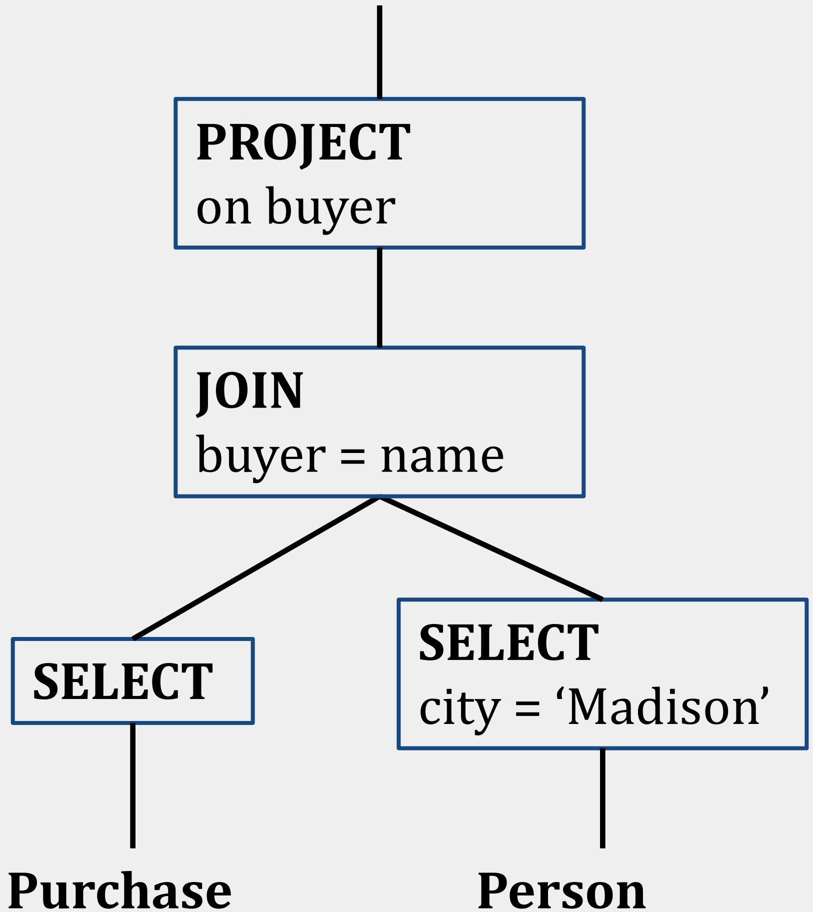
EXAMPLE QUERY

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```

- Assume that Person has a B+ tree index on city

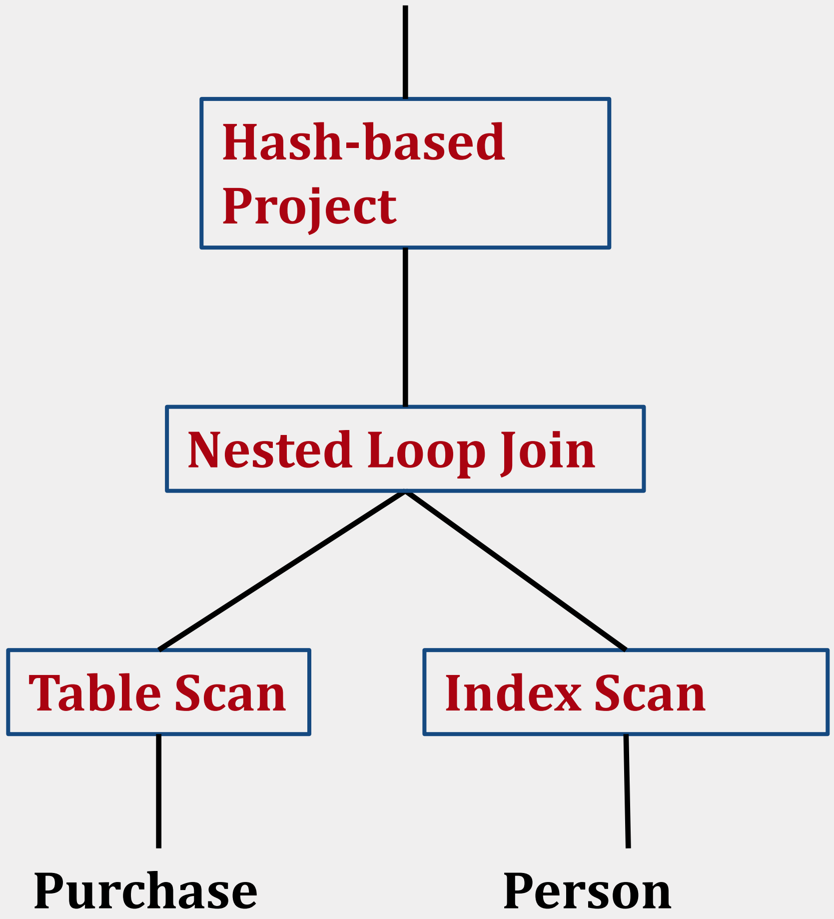
EXAMPLE: LOGICAL PLAN

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city=‘Madison’
```



EXAMPLE: PHYSICAL PLAN

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city=‘Madison’
```



SELECTION

SELECT OPERATOR

access path = way to retrieve tuples from a table

- **File Scan:**
  - scan the entire file
  - I/O cost: O(N), where N = number of pages
- **Index Scan:**
  - use an index available on some predicate
  - I/O cost: it varies depending on the index

INDEX SCAN COST

- Hash index: O(1)
  - but we can only use it with equality predicates

- B+ tree index: height – L_B + 1 + X
  - X depends on whether the index is clustered or not:
    - *unclustered*: X = # selected tuples in the worst case
    - *clustered*: X = (#selected tuples)/ (#tuples per page)
    - **optimization**: we can sort the rids by page number before we retrieve them from the unclustered index

B+ TREE SCAN EXAMPLE

- A relation with 1,000,000 records
- 100 records on a page
- 500 (key, rid) pairs on a page
- height of B+ tree = 3

<u>selectivity</u> = percentage of tuples that satisfy the selection condition

	1% selectivity	**10% selectivity**
clustered	3+100	3+1000
unclustered	3+10,000	3+100,000
unclustered + sorting	3+(~10,000)	3+(~10,000)

if we first sort, we will read at most all the pages in the B+ tree

GENERAL SELECTIONS

- So far we studied selection on a single attribute
- How do we use indexes when we have multiple selection conditions?
  - R.A = 10 **AND** R.A > 10
  - R.A = 10 **OR** R.B < 20

INDEX MATCHING

We say that an index *matches* a selection predicate if the index can be used to evaluate it

- relation **R**(A,B,C,D)
- hash index on composite key (A,B)

```
SELECT *
FROM   R
WHERE  A = 10 AND B = 5 ;
```

matches the index!

```
SELECT *
FROM   R
WHERE  A = 5 ;
```

does not match the index!

INDEX MATCHING: HASH INDEX

selection = `pred`$_1$ `AND` `pred`$_2$ `AND` …

A hash index on $(A, B, \dots)$ matches the selection condition if *all* attributes in the index search key appear in a predicate with equality (=)

EXAMPLE

relation **R**(A,B,C,D)

selection condition	**hash index on (A,B,C)**	**hash index on (B)**
A=5 **AND** B=3	no	yes
A>5 **AND** B<4	no	no
B=3	no	yes
A=5 **AND** C>10	no	no
A=5 **AND** B=3 **AND** C=1	yes	yes
A=5 **AND** B=3 **AND** C=1 **AND** D >6	yes	yes

The predicates A=5, B=3, C=1 that match the index are called **primary conjuncts**

INDEX MATCHING: B+ TREE

selection = `pred`$_1$ `AND pred`$_2$ `AND` …

A B+ tree index on $(A, B, \dots)$ matches the above selection condition if:

- the attributes in the predicates form a prefix of the search key of the B+ tree
- any operations can be used (=, < , > , …)

EXAMPLE

relation **R**(A,B,C,D)

selection condition	**B+ tree on (A,B,C)**	**B+ tree on (B,C)**
A=5 **AND** B=3	yes	yes
A>5 **AND** B<4	yes	yes
B=3	no	yes
A=5 **AND** C>10	yes	no
A=5 **AND** B=3 **AND** C=1	yes	yes
A=5 **AND** B=3 **AND** C=1 **AND** D >6	yes	yes

MORE ON INDEX MATCHING

A predicate can match *more than one* index

- hash index on (A) and B+ tree index on (B, C)
- selection: `A=7` **`AND`** `B=5` **`AND`** `C=4`

Which index should we use?

1. use the hash index, then check the conditions B=5, C=4 for every retrieved tuple
2. use the B+ tree, then check the condition A=7 for every retrieved tuple
3. use both indexes, intersect the rid sets, and only then fetch the tuples

SELECTION WITH DISJUNCTION (1)

- hash index on (A) **+** hash index on (B)
- selection: `A=7 OR B>5`

- Only the first predicate matches an index
- The only option is to do a file scan

SELECTION WITH DISJUNCTION

- hash index on (A) **+** B+ tree on (B)
- `A=7` **`OR`** `B>5`

- One solution is to do a file scan
- A second solution is to use both indexes, fetch the rids, and then do a union, and only then retrieve the tuples

Why do we need to perform the union before fetching the tuples?

SELECTION WITH DISJUNCTION

- hash index on (A) **+** B+ tree on (B)
- `(A=7 OR C>5) AND B > 5`

- We can use the B+ tree to fetch the tuples that satisfy the second predicate (B >5), then filter according to the first

CHOOSING THE RIGHT INDEX

Selectivity of an access path = *fraction* of tuples that need to be retrieved

- We want to choose the *most selective* path!
- Estimating the selectivity of an access path is generally a hard problem

ESTIMATING SELECTIVITY (1)

- selection: `A=3` **`AND`** `B=4` **`AND`** `C=5`
- hash index on (A,B,C)

The selectivity can be approximated by: ***1/#keys***

- *#keys* is known from the index
- this assumes that the values are distributed *uniformly* across the tuples

EXAMPLE

- selection: `A=3 AND B=4 AND C=5`
- *clustered* hash index on (A,B,C)
- *#pages* = 10,000
- *#keys in hash index* = 100

- selectivity = *1%*
- number of pages retrieved $= \; 10{,}000 * 1\% \; = \; 100$
- I/O cost ~ 100 + (a small constant)

ESTIMATING SELECTIVITY (2)

- selection: `A=3 AND B=4 AND C=5`
- hash index on (B,A)

If we don't know the *#keys* for the index, we can estimate selectivity as follows:

- multiply the selectivity for each primary conjunct
- If *#keys* is not known for an attribute, use 1/10 as default value
- this assumes independence of the attributes!

ESTIMATING SELECTIVITY (3)

- Selection: `A>10 AND A<60`

- If we have a range condition, we assume that the values are uniformly distributed
- The selectivity will be approximated by $\frac{interval}{High-Low}$

Example: if A takes values in $[0,100]$ then the selectivity will be $\sim \frac{60-10}{100-0} = 50\%$

PROJECTION

PROJECT OPERATOR

Simple case: `SELECT R.A, R.D`

– scan the file and for each tuple output R.A, R.D

Hard case: `SELECT DISTINCT R.A, R.D`

– project out the attributes

– eliminate *duplicate tuples* (this is the difficult part!)

PROJECT: SORT-BASED

Naïve algorithm:

1. scan the relation and project out the attributes
2. sort the resulting set of tuples using all attributes
3. scan the sorted set by comparing only adjacent tuples and discard duplicates

RUNNING EXAMPLE

`R(A,B,C,D,E)`

- $N = 1000$ pages
- $B = 20$ buffer pages
- Each field in the tuple has the same size
- Suppose we want to project on attribute `A`

SORT-BASED COST ANALYSIS

We will generally ignore the cost of writing the final result to disk, since it will be the same for every algorithm!

- initial scan = *1000 I/Os*
- after projection T =(1/5)*1000 = 200 pages
- cost of writing T = *200 I/Os*
- sorting in 2 passes = 2 * 2 * 200 = *800 I/Os*
- final scan = *200 I/Os*

total cost = 2200 I/Os

PROJECT: SORT-BASED

We can improve upon the naïve algorithm by modifying the sorting algorithm:

1. In Pass **0** of sorting, project out the attributes
2. In subsequent passes, eliminate the duplicates while merging the runs

SORT-BASED COST ANALYSIS

- we can sort in 2 passes
- pass **0** costs *1000 + 200 = 1200 I/Os*
- pass **1** costs *200 I/Os* (not counting writing the result to disk)

total cost = 1400 I/Os

PROJECT: HASH-BASED

2-phase algorithm:

- **partitioning**
  - project out attributes and split the input into B-1 partitions using a hash function *h*
- **duplicate elimination**
  - read each partition into memory and use an in-memory hash table (with a *different* hash function) to remove duplicates

PROJECT: HASH-BASED

When does the hash table fit in memory?

- size of a partition = $T \ / \ (B - 1)$, where T is #pages after projection
- size of hash table = $f \cdot T \ / \ (B - 1)$, where f is a fudge factor (typically ~ 1.2)
- So, it must be $B \ > \ f \cdot T \ / \ (B - 1)$, or approximately $B > \sqrt{f \cdot T}$

HASH-BASED COST ANALYSIS

- T = 200 so the hash table fits in memory!
- partitioning cost = 1000 + 200 = 1200 I/Os
- duplicate elimination cost = 200 I/Os

total cost = 1400 I/Os

COMPARISON

- Benefits of sort-based approach
  - better handling of skew
  - the result is sorted

- The I/O costs are the same if $B^2 > T$
  - 2 passes are needed by both algorithms

PROJECT: INDEX-BASED

- Index-only scan
  - projection attributes subset of index attributes
  - apply projection algorithm only to data entries
- If an *ordered index* contains all projection attributes as prefix of search key:
  1. retrieve index data entries in order
  2. discard unwanted fields
  3. compare adjacent entries to eliminate duplicates

Relational Operators #2

CS 564- Spring 2020

ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

Algorithms for relational operators:

- joins
- set operators
- aggregation

JOINS

JOIN OPERATOR

Algorithms for equijoin:

```
SELECT *
FROM   R, S
WHERE  R.a = S.a
```

Why can't we compute it as cartesian product?

JOIN ALGORITHMS

Algorithms for equijoin:

- nested loop join
- block nested loop join
- index nested loop join
- block index nested loop join
- sort merge join
- hash join

NESTED LOOP JOIN (1)

- for each page P_R in **R**
  - for each page P_S in **S**
    - join the tuples on P_R with the tuples in P_S

I/O = $M_R + M_S \cdot M_R$

- M_R = number of pages in **R**
- M_S = number of pages in **S**

Observe that we ignore the cost of writing the output to disk!

NESTED LOOP JOIN (2)

- Which relation should be the outer relation in the loop?
  - The smaller of the two relations
- How many buffer pages do we need?
  - only 3 pages suffice

BLOCK NESTED LOOP JOIN (1)

Assume B buffer pages

- for each block of B-2 pages from **R**
  - for each page P_S in **S**
    - join the tuples from the block with the tuples in P_S

$$\text{I/O} = M_R + M_S \cdot \left\lceil \frac{M_R}{B-2} \right\rceil$$

BLOCK NESTED LOOP JOIN (2)

- To increase CPU efficiency, create an in-memory hash table for each block
  - what will be the key of the hash table?

- What happens if **R** fits in memory?
  - The I/O cost is only $M_R + M_S$!

NLJ VS BNLJ

Example:

- M_R = 500 pages
- M_S = 1000 pages
- 100 tuples / page
- $B = 12$

NLJ I/O = 500 + 500*1,000 = **500,500**

BNLJ I/O = 500 + $\frac{500*1,000}{12-2}$ = **50,500**

The difference in I/O cost in an order of magnitude!

INDEX NESTED LOOP JOIN

S has an index on the join attribute

- for each page P_R in **R**
  - for each tuple r in **R**
    - probe the index of **S** to retrieve any matching tuples

I/O = $M_R + |R| \cdot I^*$

- I^* is the I/O cost of searching an index, and depends on the type of index and whether it is clustered or not

BLOCK INDEX NESTED LOOP JOIN

- for each block of B-2 pages in **R**
  - sort the tuples in the block
  - for each tuple r in the block
    - probe the index of **S** to retrieve any matching tuples

- Why do we need to sort here?

SORT MERGE JOIN

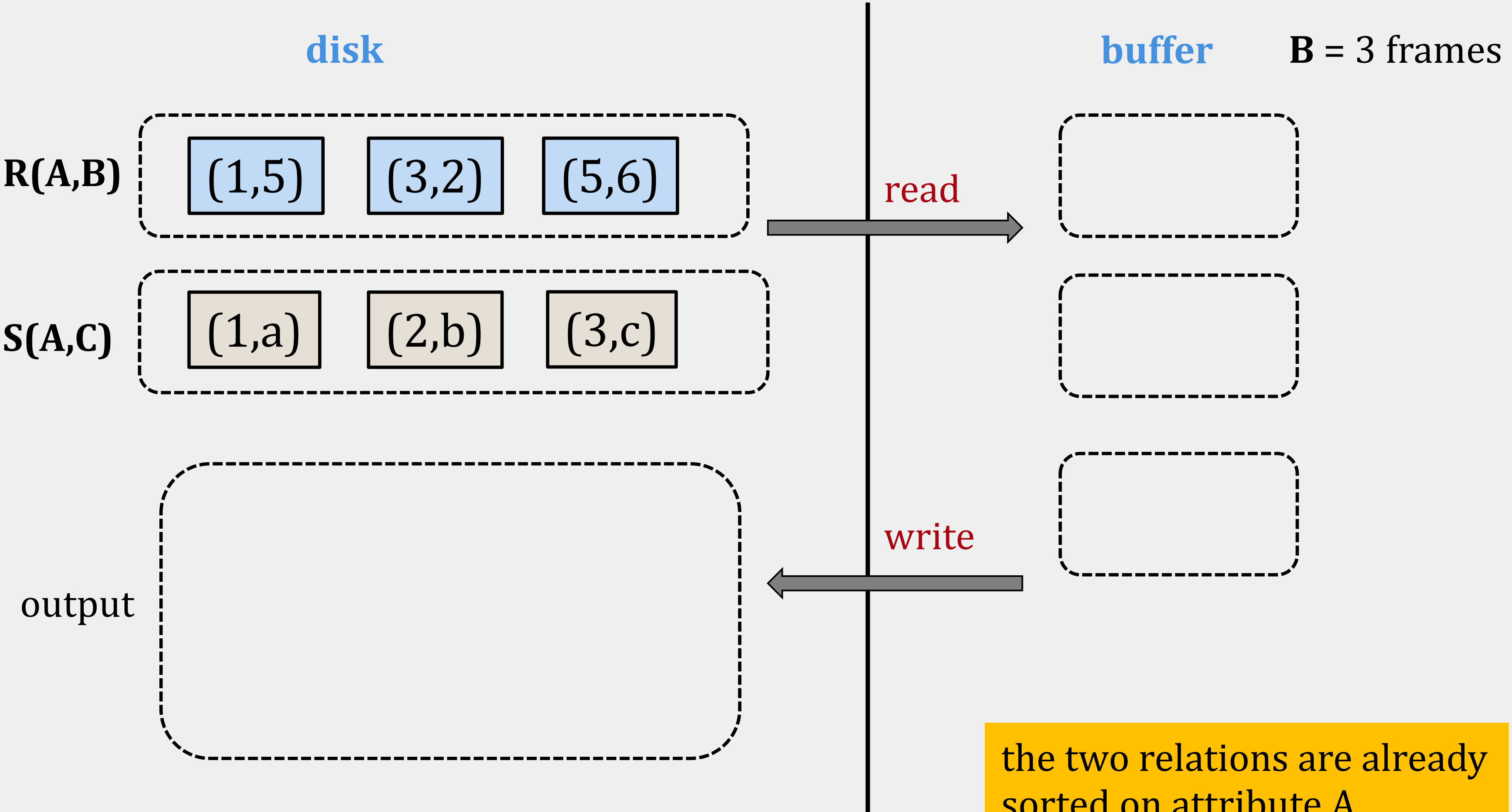
SORT MERGE JOIN: BASIC VERSION

The basic version:

- sort **R** and ***S*** on the join attribute (using external merge sort)
- read the sorted relations in the buffer and merge

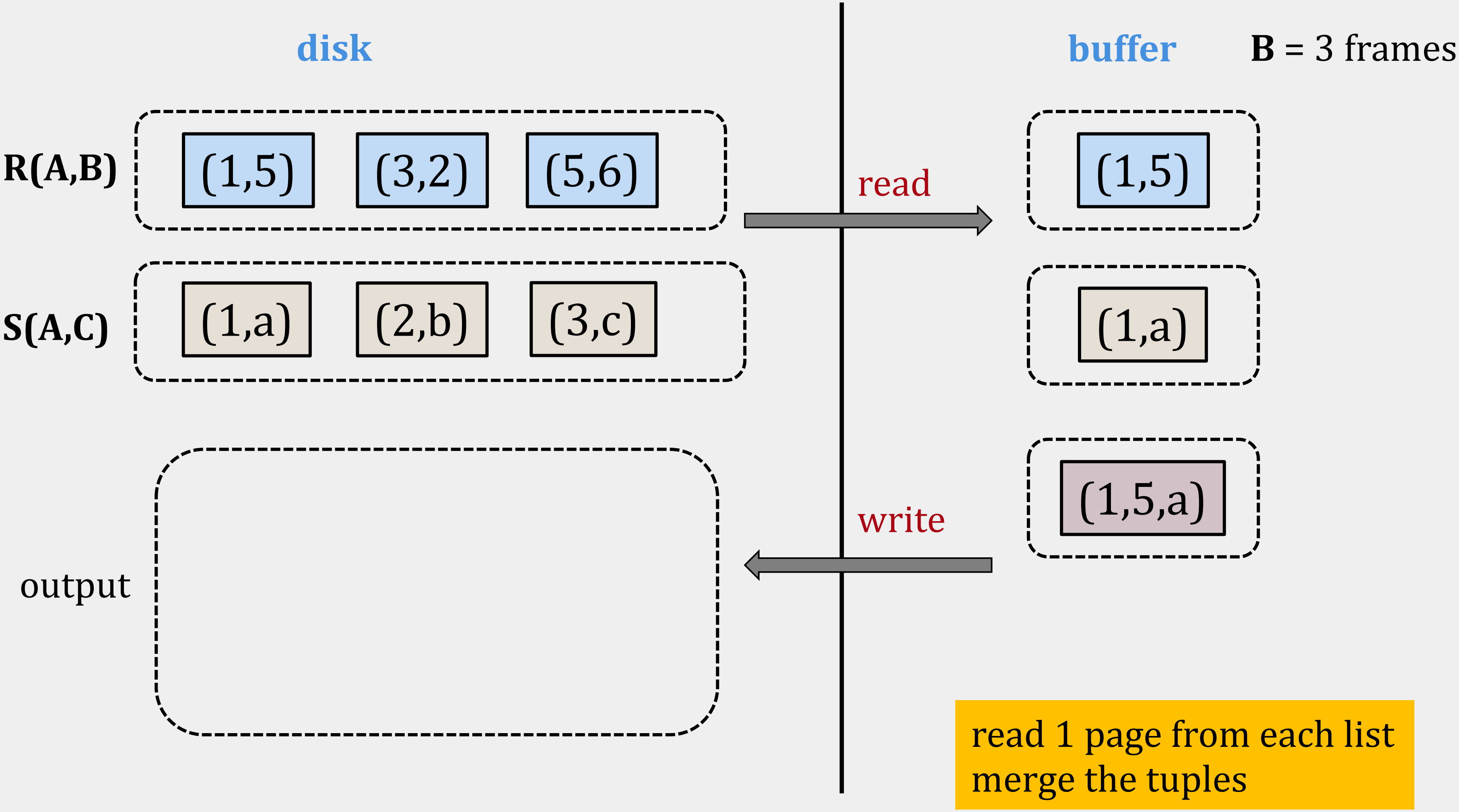
If **R**, **S** are already sorted on the join attribute we can skip the first step!

READ AND MERGE

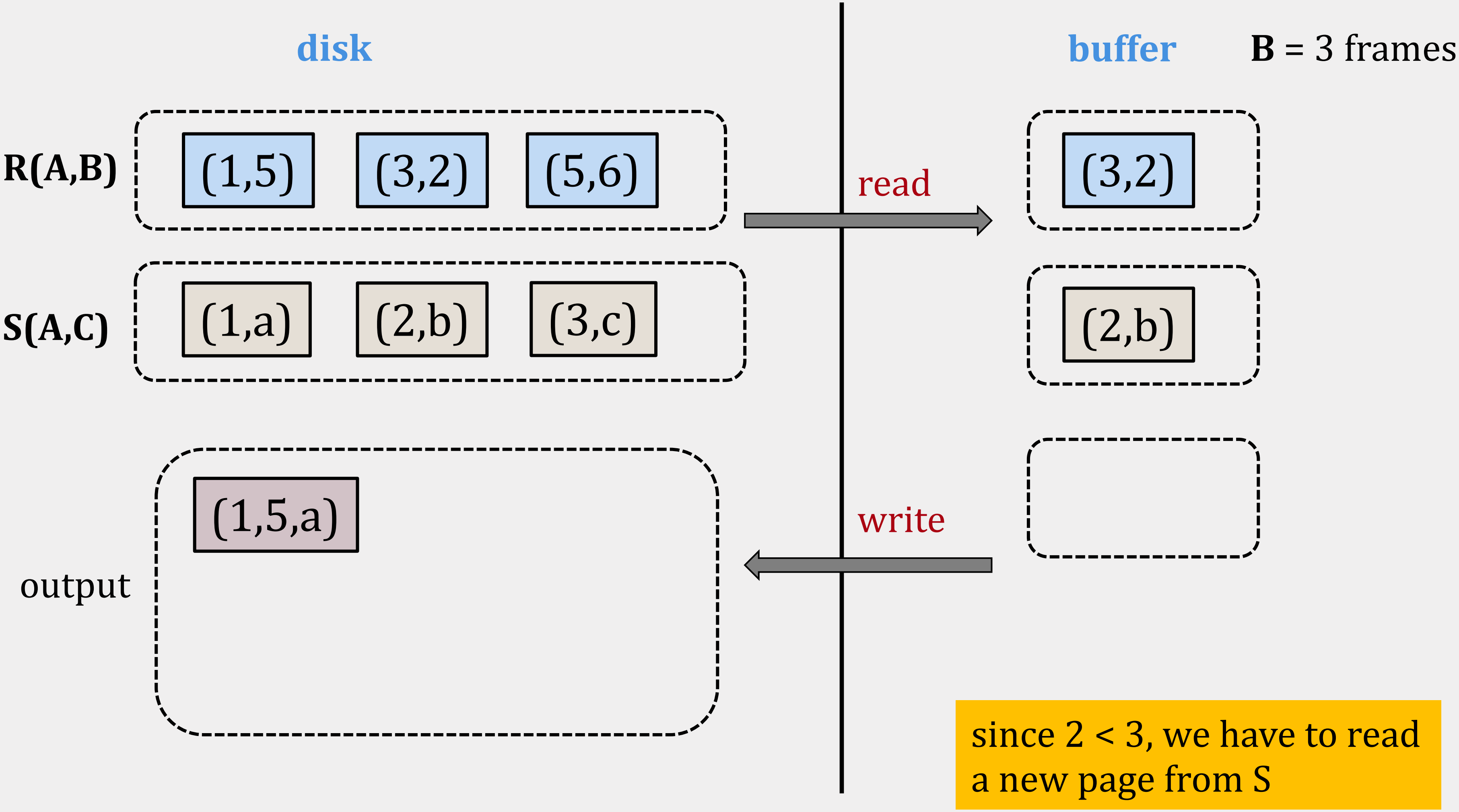


the two relations are already sorted on attribute A

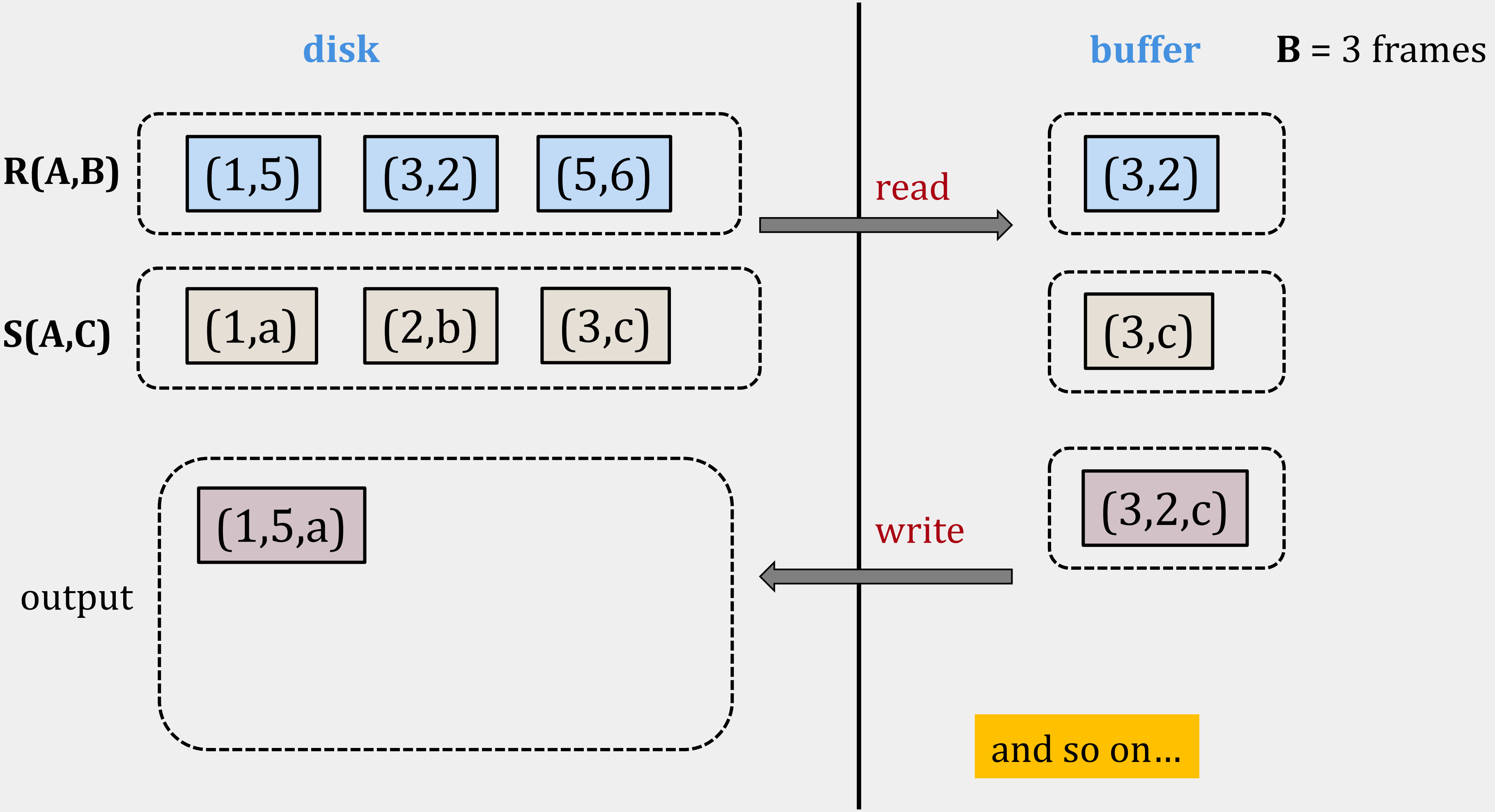
READ AND MERGE



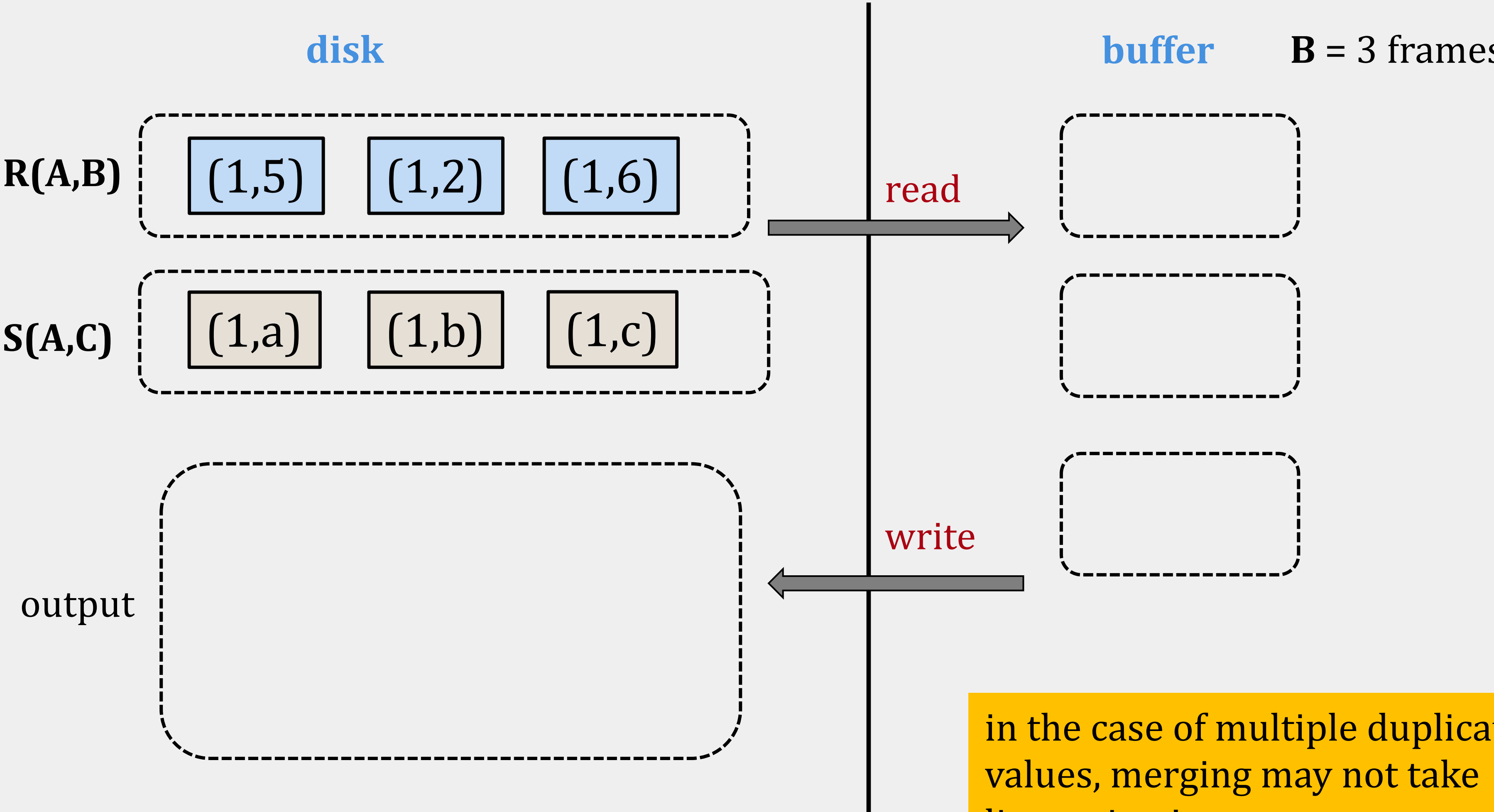
READ AND MERGE



READ AND MERGE

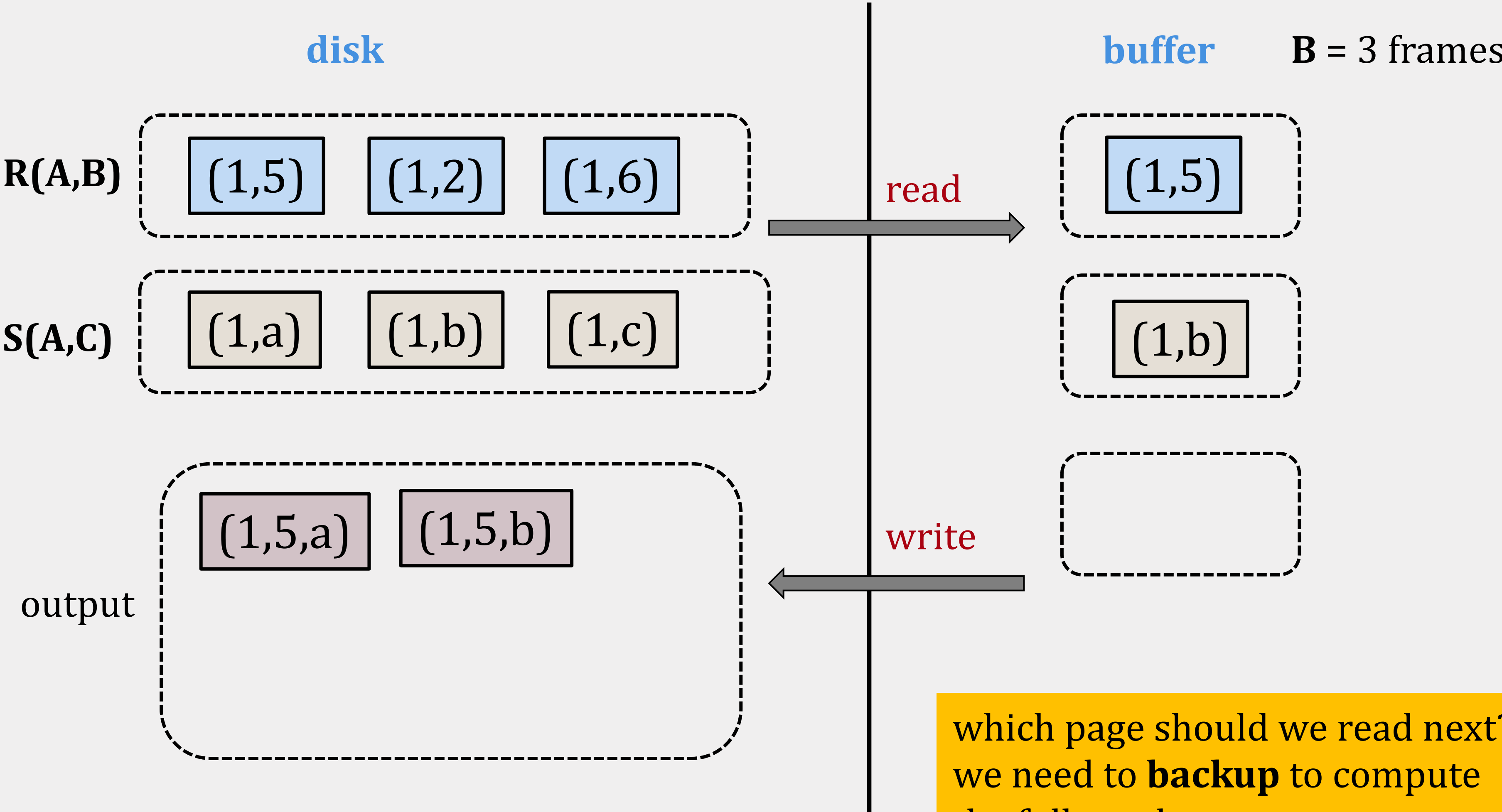


SORTING WITH DUPLICATES



in the case of multiple duplicate values, merging may not take linear time!

SORTING WITH DUPLICATES



which page should we read next? we need to **backup** to compute the full result

SMJ: I/O COST

- If there is no backup, the I/O cost of read + merge is only $M_R + M_S$
- If there is backup, in the worst case the I/O cost could be $M_R * M_S$
  - this happens when there is a *single* join value

Total I/O cost ~ $sort(R) + sort(S) + M_R + M_S$

SORT MERGE JOIN: OPTIMIZED

- Generate sorted runs of size $\sim 2B$ for **R** and **S**
- Merge the sorted runs for **R** and **S**
  - while merging check for the join condition and output the join tuples

I/O cost $\sim 3(M_R + M_S)$

But how much memory do we need for this to happen?

SMJ: MEMORY ANALYSIS

- In the first phase, we create runs of length ~2B
- Hence, the number of runs is $\frac{M_R+M_S}{2B}$
- To perform a k-way merge, we need k+1 buffer pages, so:

$$\frac{M_R+M_S}{2B} \leq B-1 \text{ or } B^2 \geq \max\{M_S, M_R\}$$

If B^2 is larger than the **maximum** number of pages of the two relations, then SMJ has I/O cost ~ $3(M_R + M_S)$

HASH JOIN

HASH FUNCTION REFRESHER

- We will use a hash function *h* to map values of the join attribute (A) into buckets [1, B-1]
- Tuple *t* is then hashed to bucket *h(t.A)*

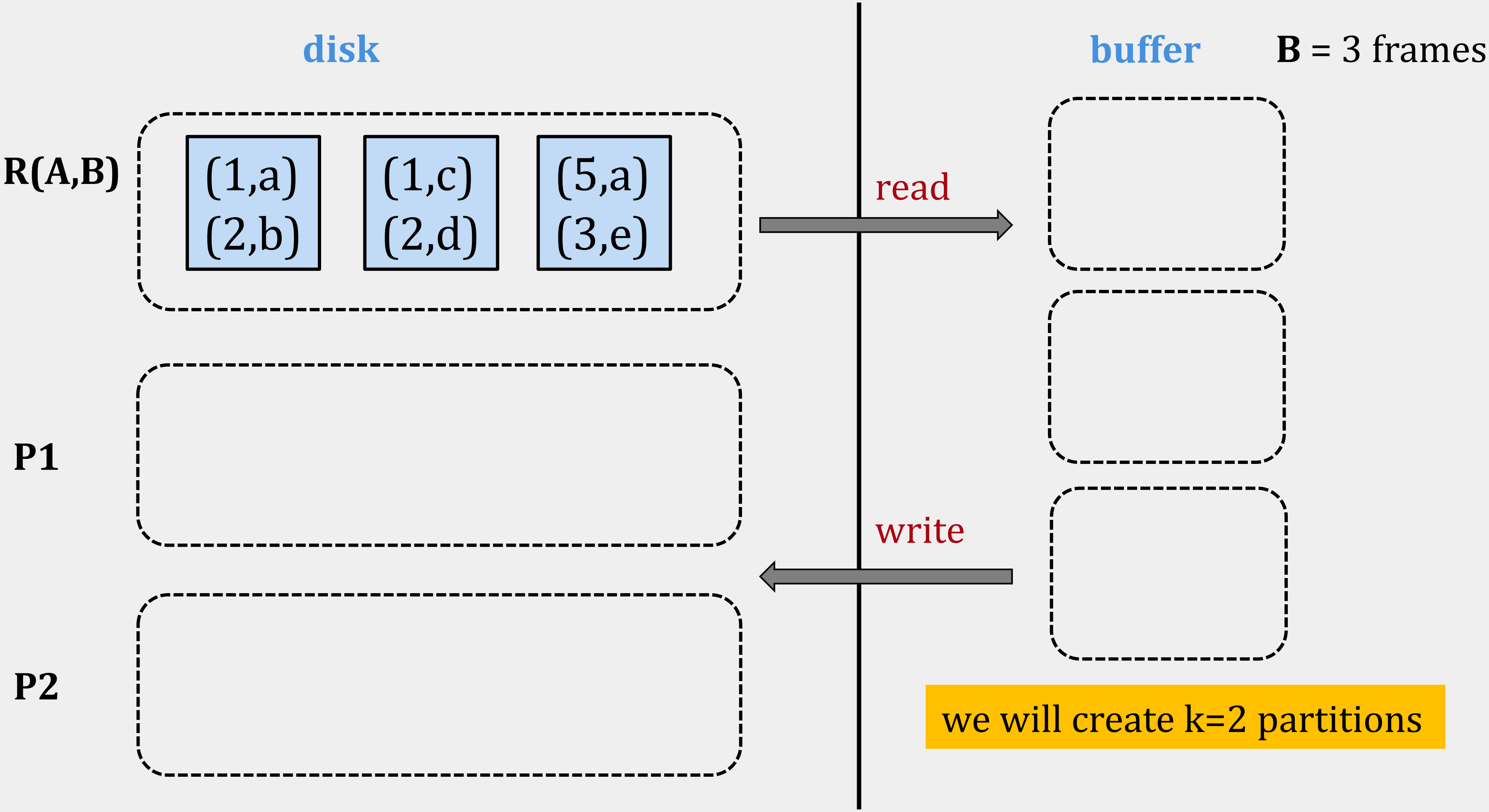
- A hash **collision** occurs when x != y but h(x) = h(y)
- Note however that it will **never** happen that x = y but h(x) != h(y)

HASH JOIN: OVERVIEW

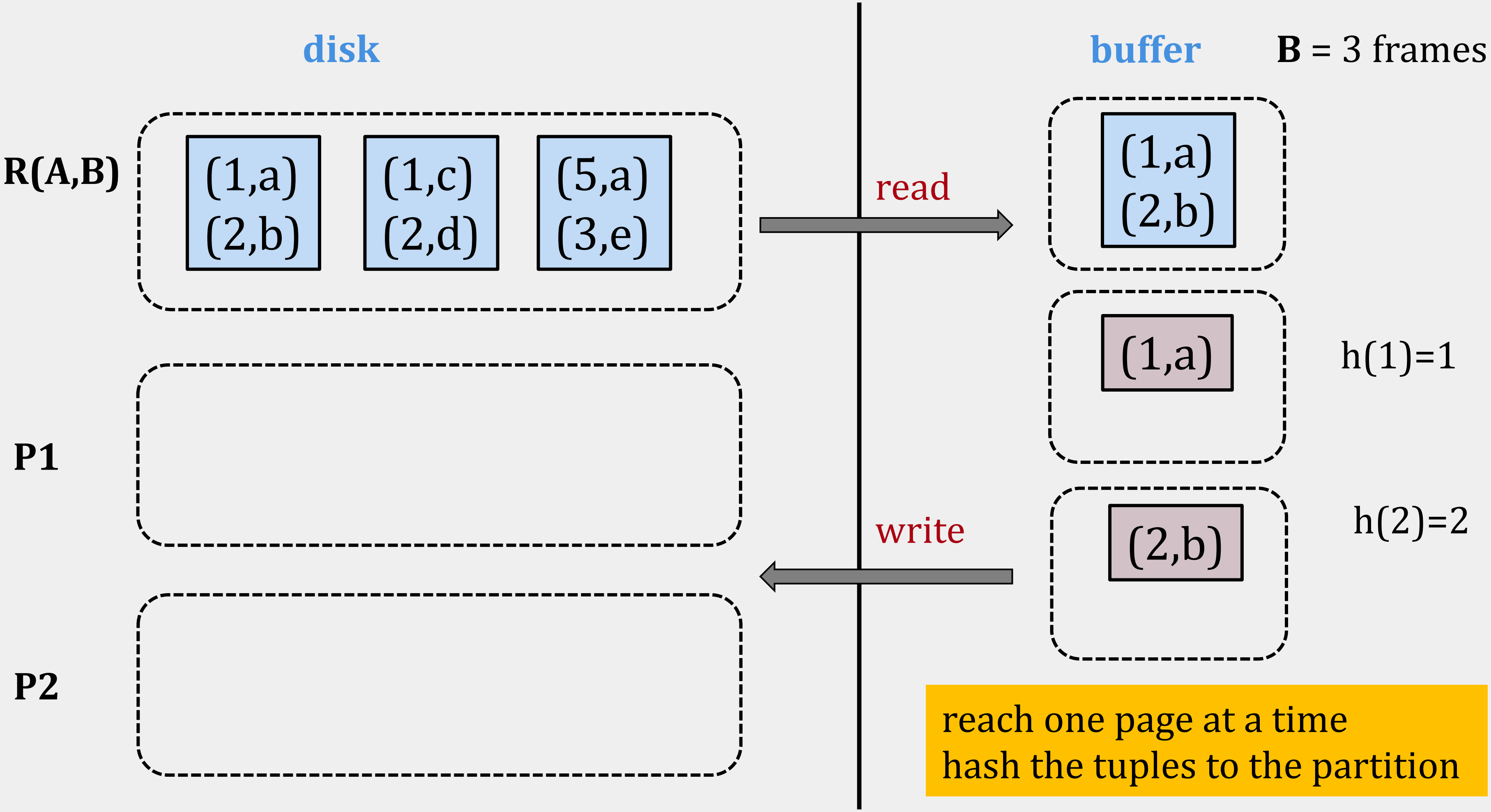
Start with a hash function h on the join attribute

- **Partition phase:** partition **R** and **S** into k partitions using h
- **Matching phase:** join each partition of **R** with the corresponding (same hash value) partition of **S** using BNLJ

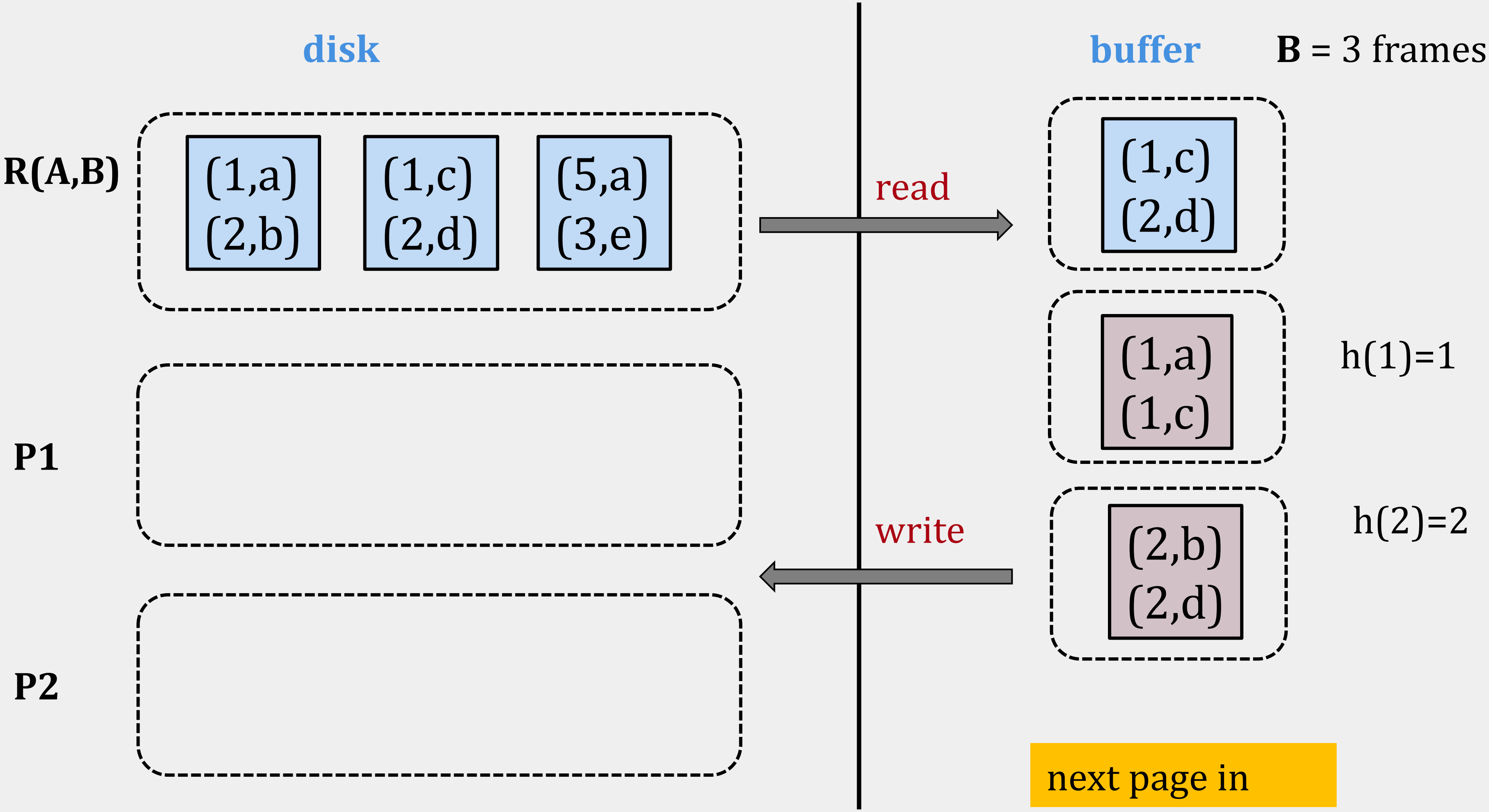
PARTITION PHASE



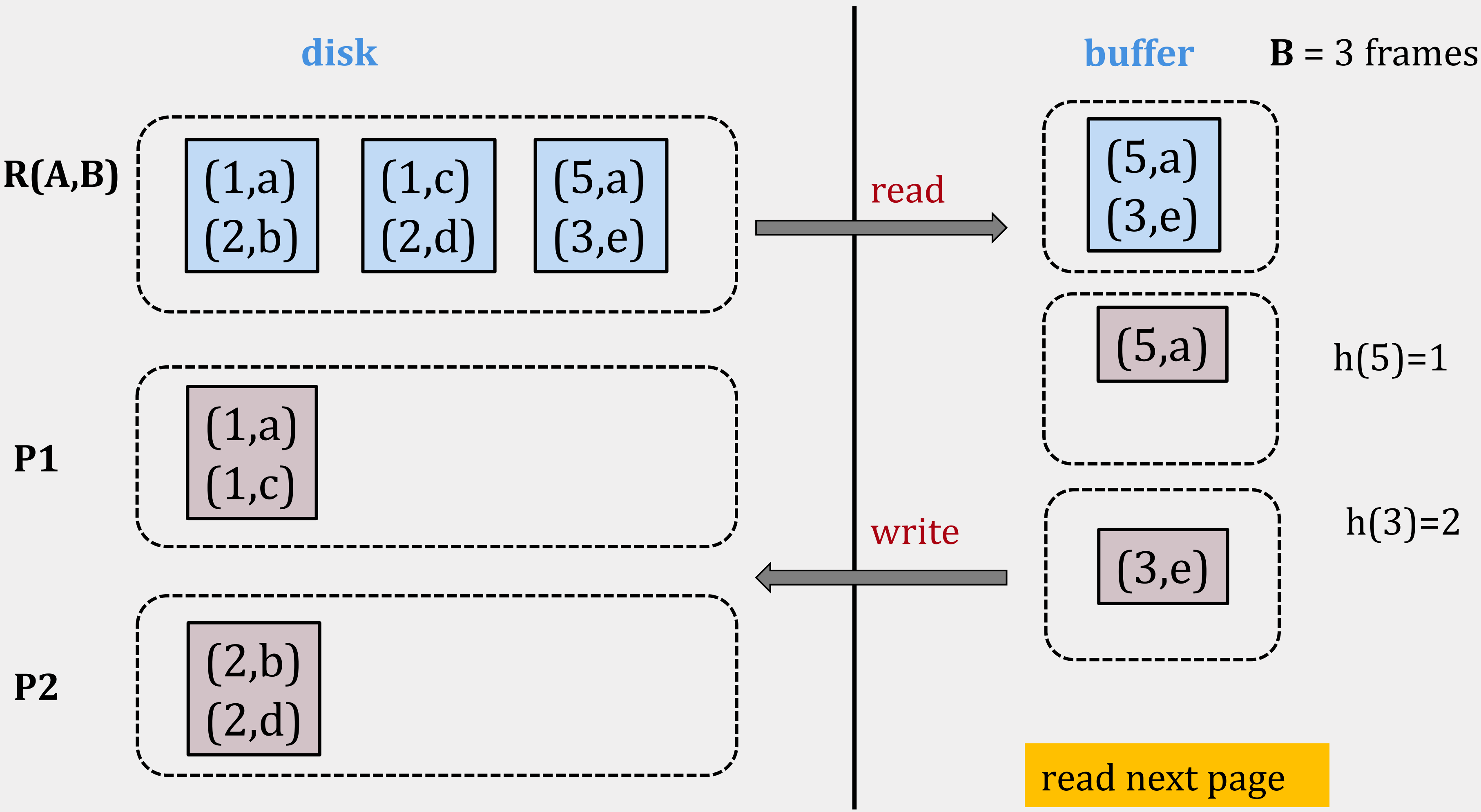
PARTITION PHASE



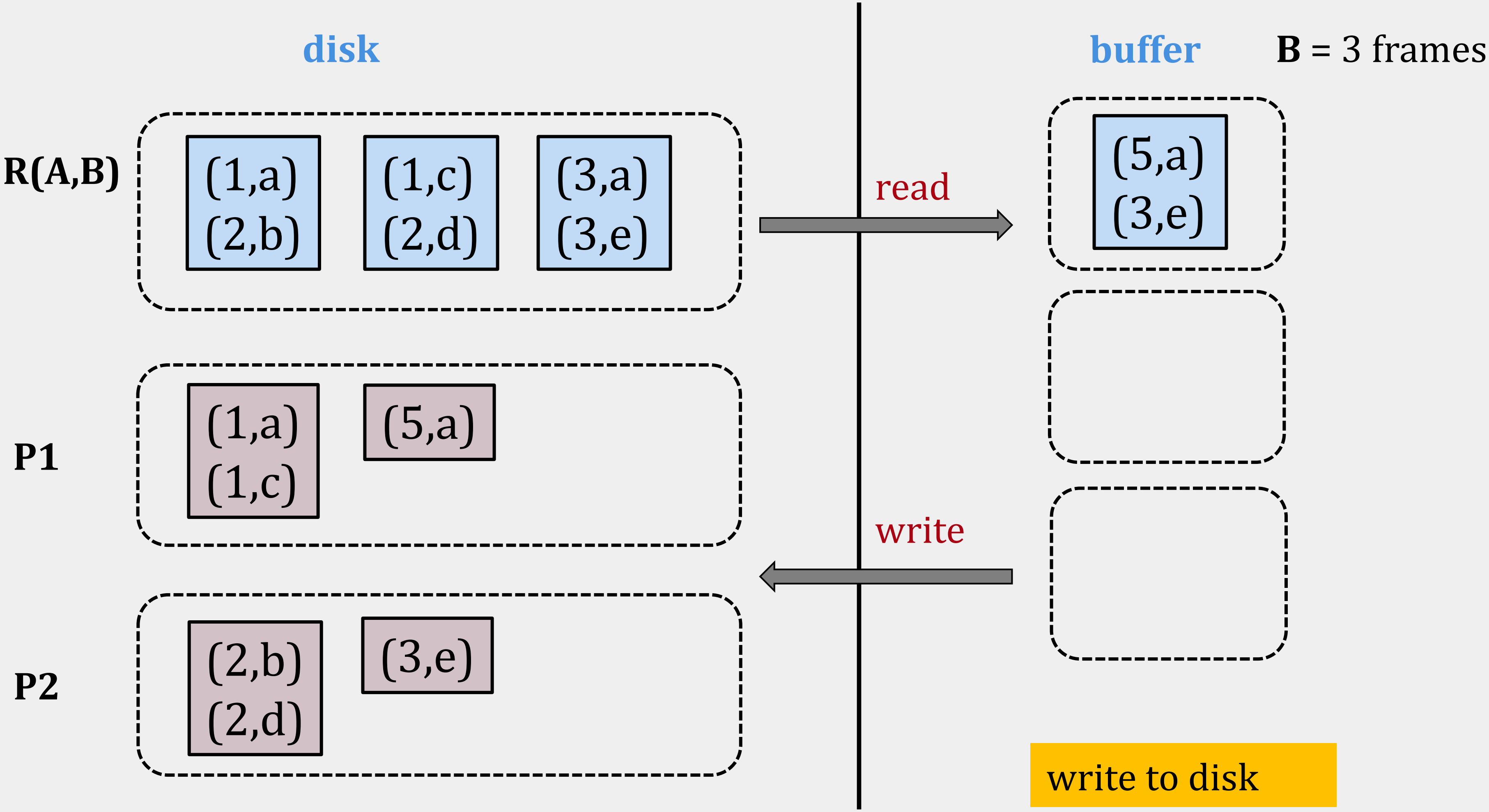
PARTITION PHASE



PARTITION PHASE



PARTITION PHASE



BUCKET SIZE

- We can create up to k = B-1 partitions in one pass
- How big are the buckets we create?
  - Ideally, each bucket has ~ M/(B-1) pages
  - but hash collisions can occur!
  - or we may have many duplicate values on the join attribute (skew)

- In the matching phase, we join two buckets from R, S with the same hash value
  - We want to do this in linear time using BNLJ, so we must guarantee that each bucket from one of the two relations is at most B-1 pages

HJ: I/O COST

- Suppose $M_R \leq M_S$
- The partition phase gives buckets of size $\sim M_R/B$
- To make BNLJ run in one pass we need to make sure that:

$$\frac{M_R}{B} \leq B - 2 \text{ or equivalently: } B^2 \geq M_R$$

If B^2 is larger than the **minimum** number of pages of the two relations, then HJ has I/O cost $\sim 3(M_R + M_S)$

COMPARISON OF JOIN ALGORITHMS

Hash Join **vs** Block Nested Loop Join

- the same if smaller table fits into memory
- otherwise, hash join is much better

COMPARISON OF JOIN ALGORITHMS

Hash Join **vs** Sort Merge Join

- Suppose $M_R > M_S$
- To do a two-pass join, SMJ needs $B > \sqrt{M_R}$
  – the I/O cost is: $3(M_R + M_S)$
- To do a two-pass join, HJ needs $B > \sqrt{M_S}$
  – the I/O cost is: $3(M_R + M_S)$

GENERAL JOIN CONDITIONS

- Equalities over multiple attributes
  - e.g., *R.sid=S.sid* **and** *R.rname=S.sname*
  - for Index Nested Loop
    - index on <*sid, sname*>
    - index on *sid* or *sname*
  - for SMJ and HJ, we can sort/hash on combination of join attributes

GENERAL JOIN CONDITIONS

- Inequality conditions
  - e.g., *R.rname < S.sname*
  - For BINL, we need (clustered) B+ tree index
  - SMJ and HJ not applicable
  - BNLJ likely to be the winner (why?)

SET OPERATIONS & AGGREGATION

SET OPERATIONS

- Intersection is a special case of a join
- Union and difference are similar
- Sorting:
  - sort both relations (on *all attributes*)
  - merge sorted relations eliminating duplicates
- Hashing:
  - partition R and S
  - build in-memory hash table for partition R_i
  - probe with tuples in S_i, add to table if not a duplicate

AGGREGATION: SORTING

- sort on group by attributes (if any)
- scan sorted tuples, computing running aggregate
  - max/min: max/min
  - average: sum, count
- when the group by attribute changes, output aggregate result
- **cost** = sorting cost

AGGREGATION: HASHING

- Hash on group by attributes (if any)
  - Hash entry = group attributes + running aggregate
- Scan tuples, probe hash table, update hash entry
- Scan hash table, and output each hash entry
- **cost** = scan relation
- What happens if we have many groups?

AGGREGATION: INDEX

- Without grouping
  - Can use B+ tree on aggregate attribute(s)
- With grouping
  - B+ tree on all attributes in SELECT, WHERE and GROUP BY clauses
    - Index-only scan
    - If group-by attributes prefix of search key, the data entries/tuples are retrieved in group-by order

Query Optimization

CS 564- Spring 2020

ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

- What is a query optimizer?
- Generating query plans
- Cost estimation of query plans

ARCHITECTURE OF AN OPTIMIZER

query (**SQL**)

Query Parser

parsed query (**Relational Algebra**)

Query Optimizer
- Plan generator
- Plan cost estimator

System Catalog

query plan (**annotated RA tree**)

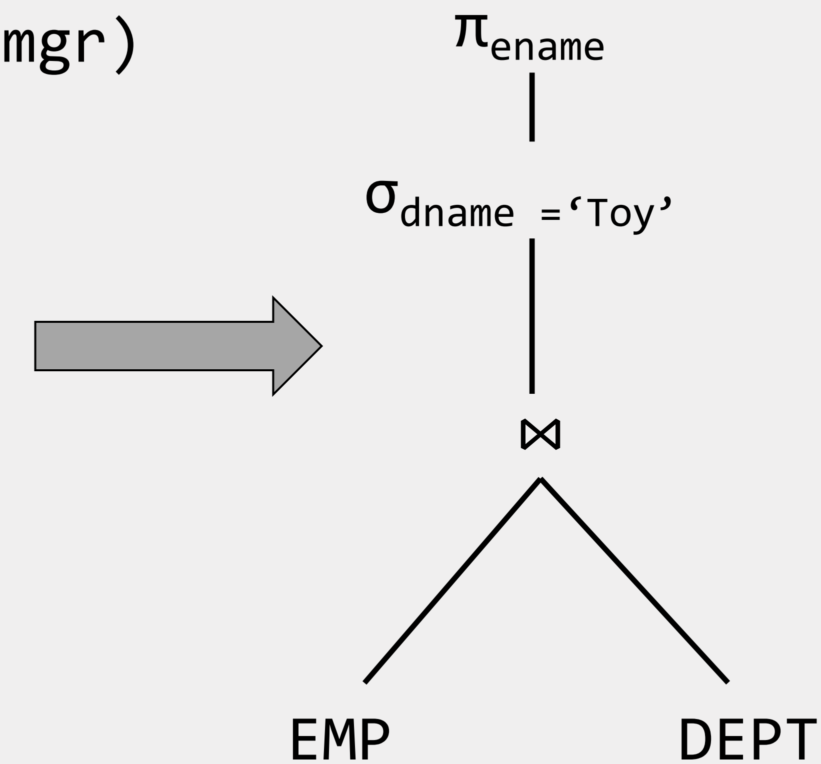
Relational Algebra is the glue!

EXAMPLE: FROM SQL TO RA

EMP(ssn, ename, addr, sal, did)

DEPT(did, dname, floor, mgr)

```
SELECT DISTINCT ename
FROM   Emp E, Dept D
WHERE  E.did = D.did
AND    D.dname = 'Toy' ;
```



QUERY OPTIMIZATION: BASICS

The query optimizer

1. identifies candidate equivalent RA trees
2. for each RA tree, it finds the best annotated version (using any available indexes)
3. chooses the best overall plan by estimating the I/O cost of each plan

Generating Query Plans

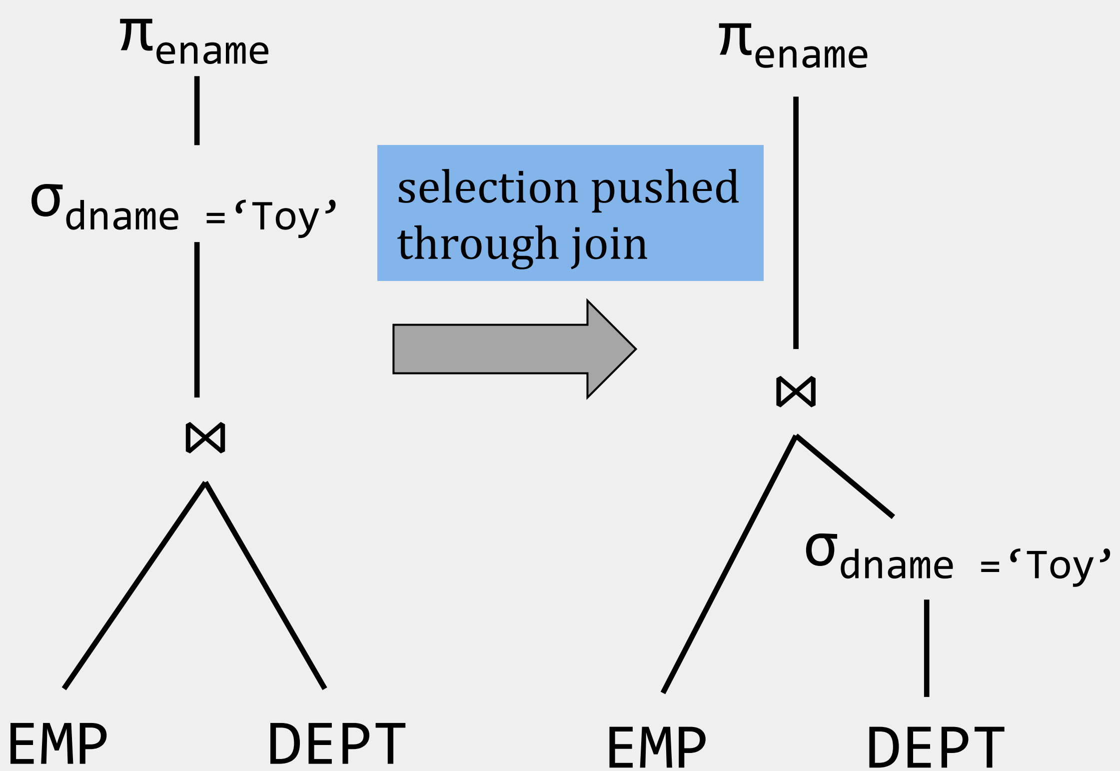
QUERY PLANS

- The space of possible query plans is typically huge and it is hard to navigate through
- Relational Algebra provides us with mathematical rules that transform one RA expression to an equivalent one
  – push down selections & projections
  – join reordering
- These transformations allow us to construct many alternative query plans

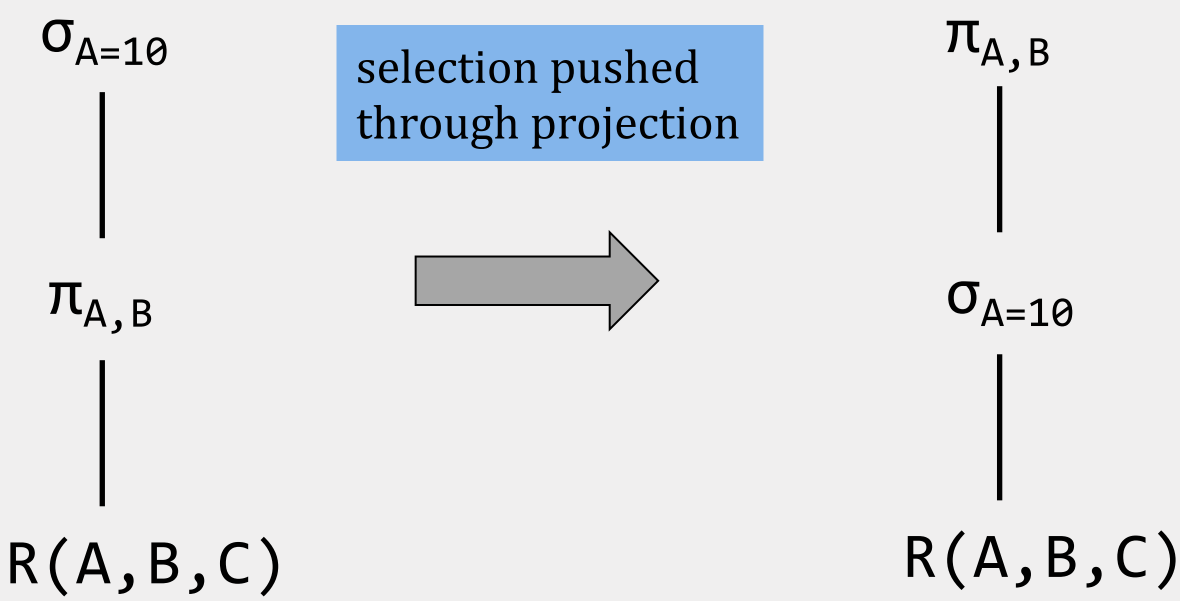
PUSHING DOWN SELECTIONS

A selection can be pushed down through

- projections
- joins
- other selections

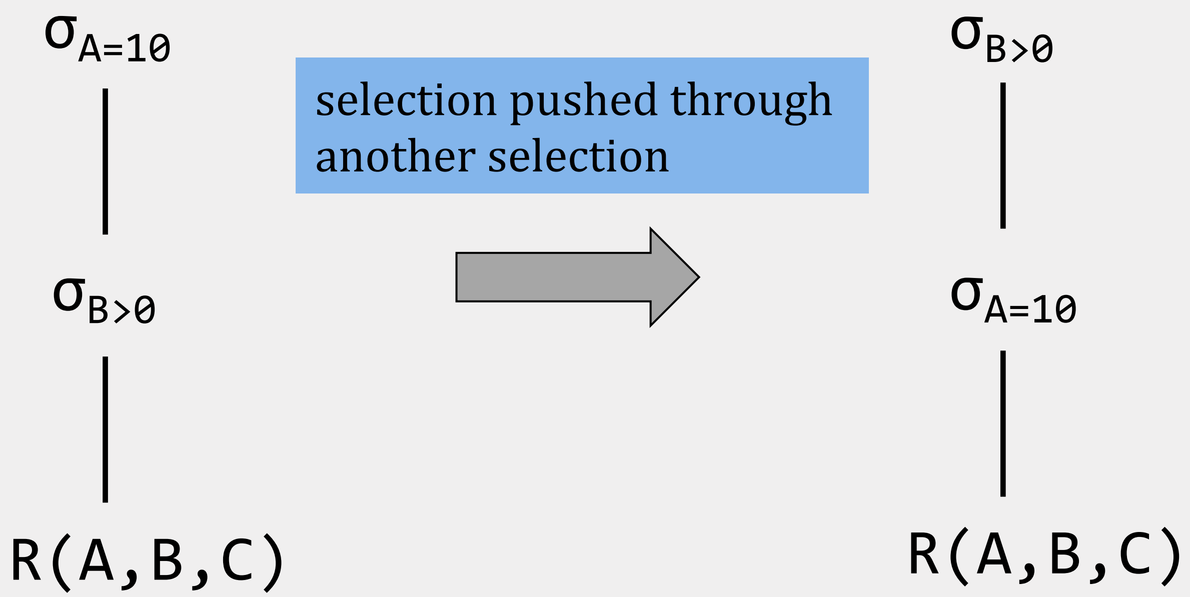


PUSHING DOWN SELECTIONS



SELECTION REORDERING

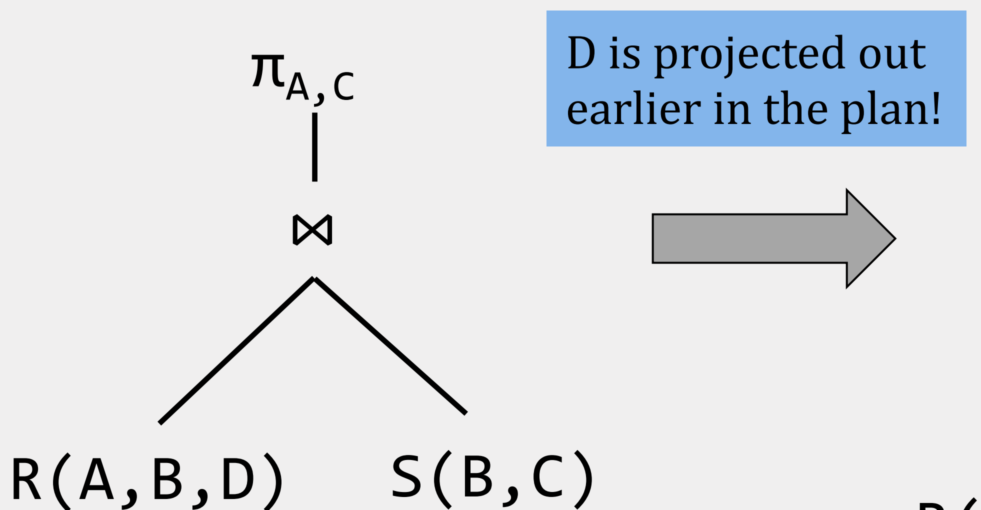
It is always possible to change the order of selections

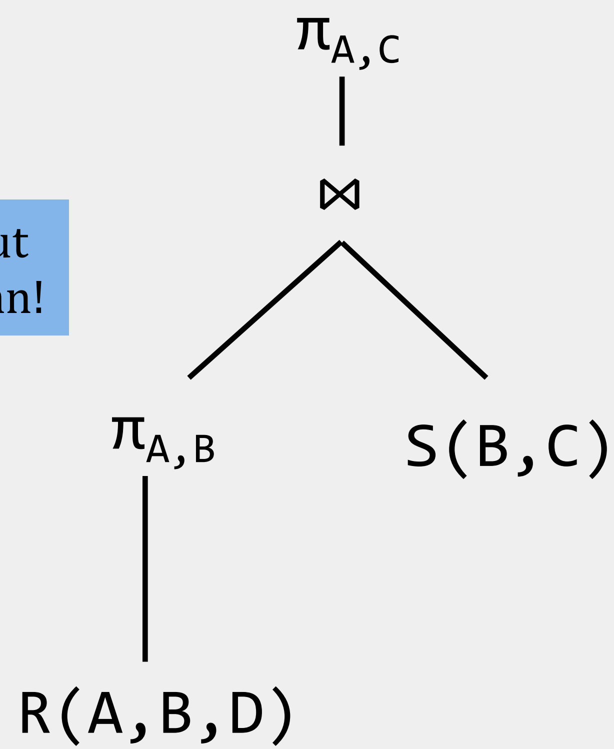


PUSHING DOWN PROJECTIONS

A projection can be pushed down through

- selections
- joins

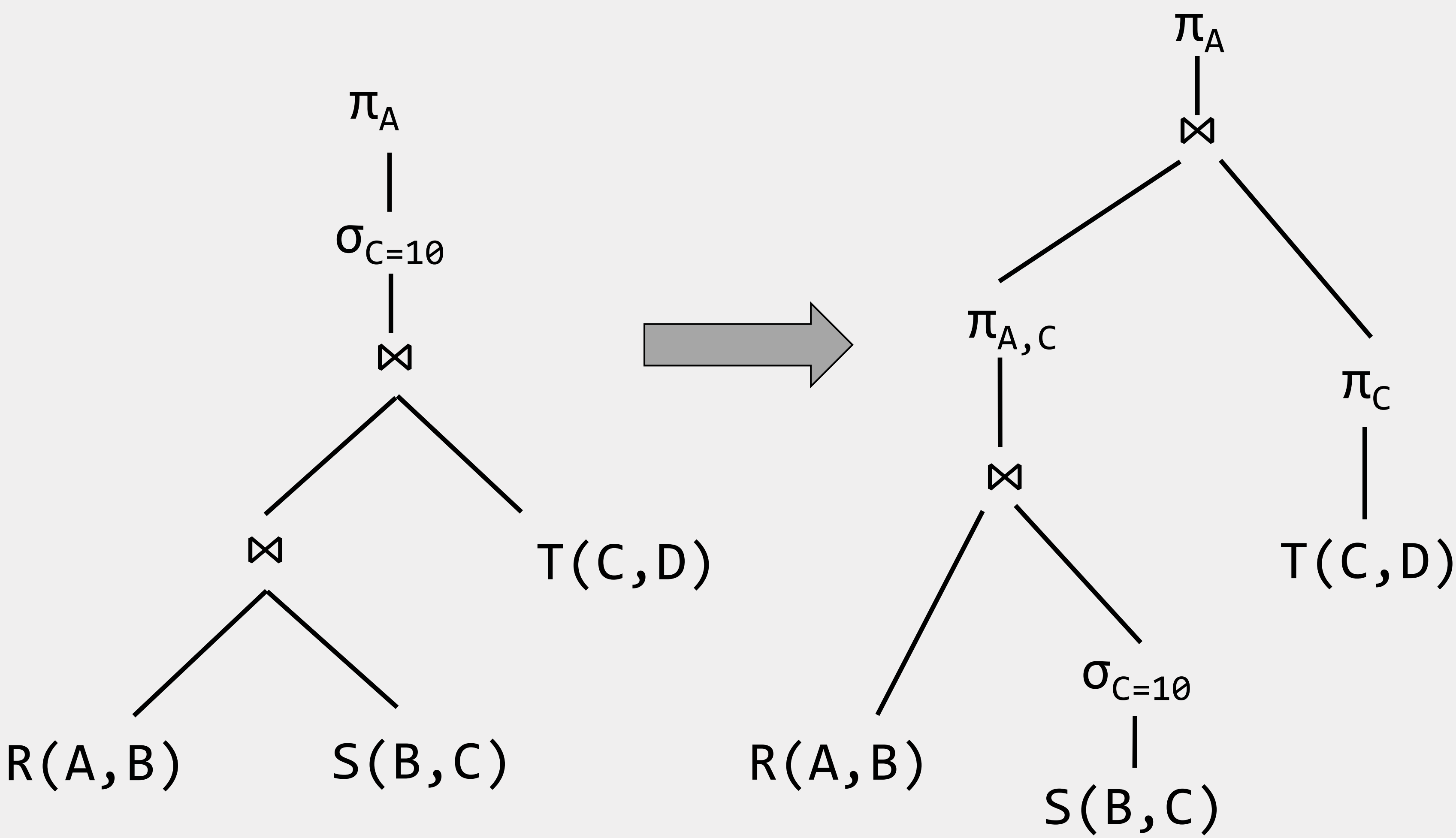




SELECTIONS & PROJECTIONS

- Heuristically, we want selections and projections to occur as early as possible in the query plan
- **The reason**: we will have fewer tuples in the intermediate steps of the plan
  - this could fail if the selection condition is very very expensive
  - projection could be a waste of effort, but more rarely

EXAMPLE



JOIN REORDERING

- Commutativity of join

  $R \bowtie S \equiv S \bowtie R$

- Associativity of join

  $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$

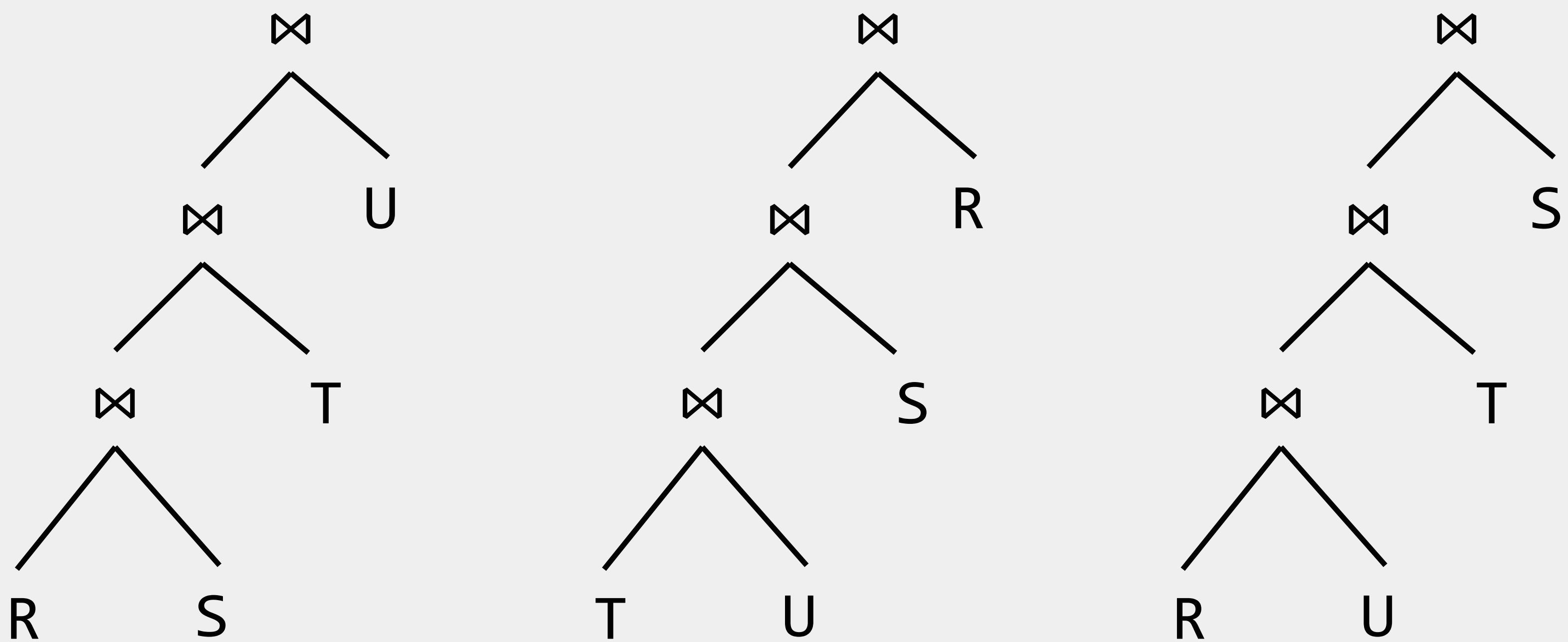
We can reorder the computation of joins in any way (exponentially many orders)!

JOIN REORDERING

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D) \bowtie U(D,E)$$

left-deep join plans

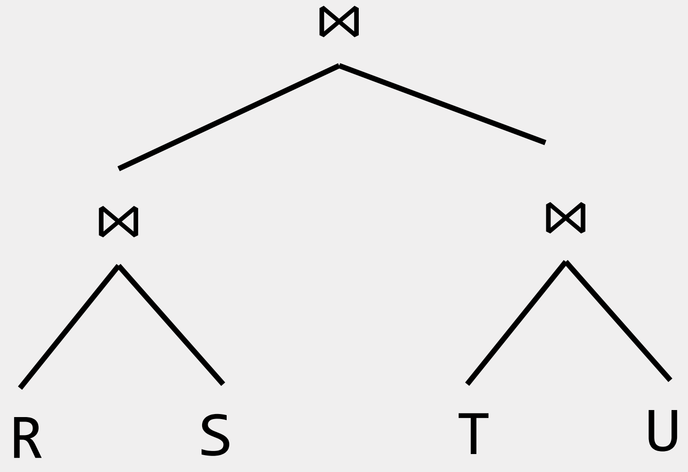
correct, but not a good plan!



JOIN REORDERING

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D) \bowtie U(D,E)$$

bushy plan



PLAN GENERATION: RECAP

- selections can be evaluated in any order
- joins can be evaluated in any order
- selections and projections can be pushed down the tree using the RA equivalence transformations

QUERY PLAN COST ESTIMATION

COST ESTIMATION

Estimating the cost of a query plan involves:

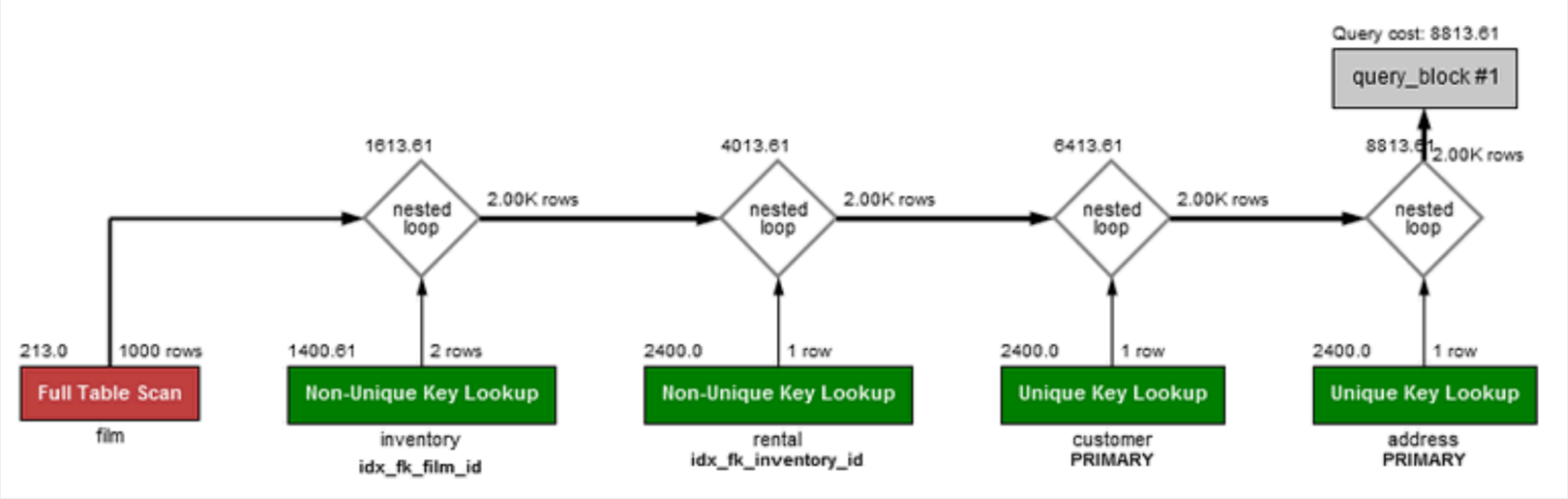
- estimating the cost of each operation in the plan
  - depends on input cardinalities
  - algorithm cost (we have seen this!)
- estimating the size of intermediate results
  - we need statistics about input relations
  - for selections and joins, we typically assume independence of predicates

COST ESTIMATION

- Statistics are stored in the system catalog:
  - number of tuples (*cardinality*)
  - size in pages
  - # distinct keys (when there is an index on the attribute)
  - range (for numeric values)
- The system catalog is updated periodically
- Commercial systems use additional statistics, which provide more accurate estimates:
  - histograms
  - wavelets

REAL-WORLD EXAMPLE

```
SELECT CONCAT(customer.last_name, ', ', customer.first_name) AS customer,
        address.phone, film.title
FROM rental
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE() LIMIT 5;
```



EXAMPLE: COST ESTIMATION

- `EMP(ssn, ename, addr, sal, did)`
  - 10000 tuples, 1000 pages
- `DEPT(did, dname, floor, mgr)`
  - 500 tuples, 50 pages
  - 100 distinct values for `dname`

```
SELECT DISTINCT ename
FROM   Emp E, Dept D
WHERE  E.did = D.did
AND    D.dname = 'Toy' ;
```

EXAMPLE: COST ESTIMATION

buffer size B= 40

cost of projection = 20

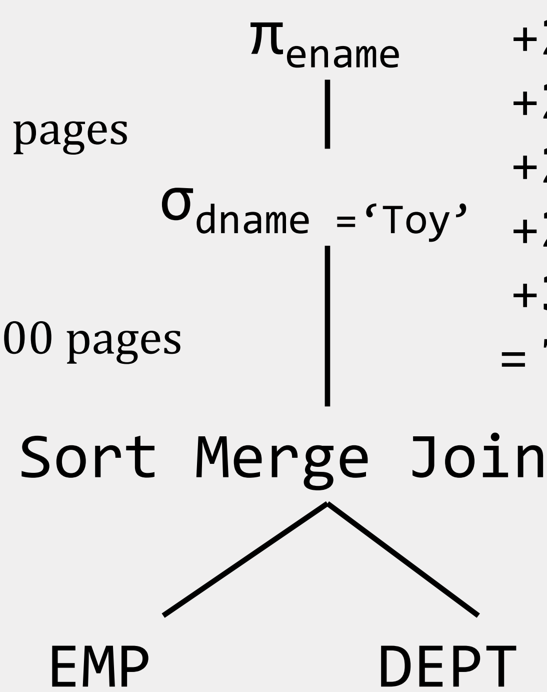
intermediate result ~ 20 pages

cost of selection = 2000

intermediate result ~ 2000 pages

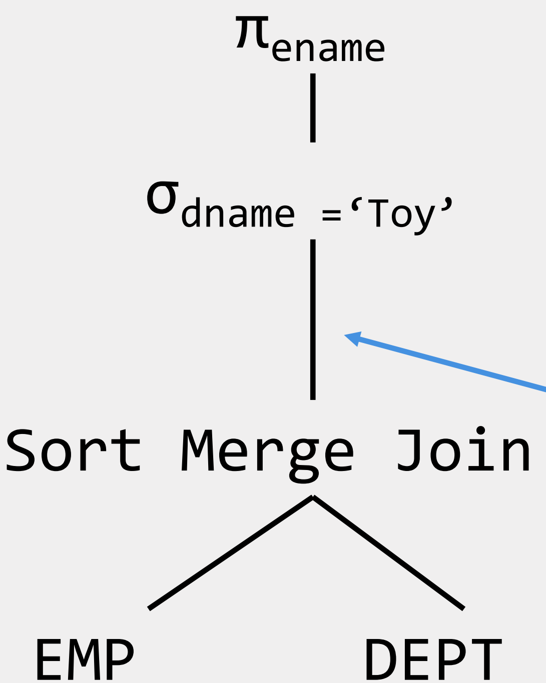
cost of SMJ = $3 * (1000 + 50)$

after each operator, we write (**materialize**) the result to disk



total I/O cost =
OUT
+20
+20 {materialize}
+2000
+2000 {materialize}
+3150
= 7550 + OUT

PIPELINING



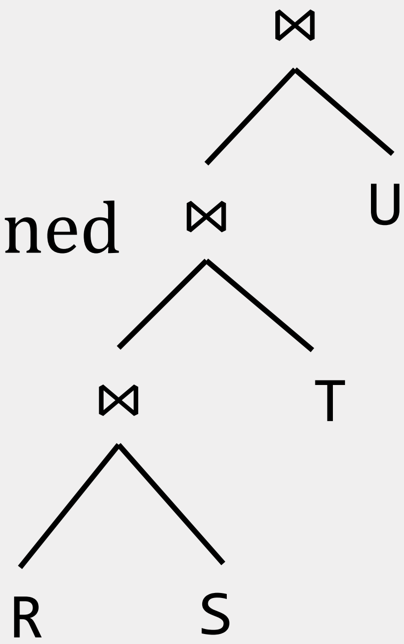
After each operator, we have 2 choices:

- materialize the intermediate result before we start the next operator
- pipeline the result to the next operator without writing to disk!

We can apply the selection condition as the tuples are generated from the join operator, before writing the full result to disk!

PIPELINING

- By using pipelining we benefit from:
  - no reading/writing to disk of the temporary relation
  - overlapping execution of operators
- Pipelining is not always possible!
- Left-deep join plans allow for fully pipelined evaluation!



for BNLJ, left child = outer relation

COST ESTIMATION W/ PIPELINING

buffer size B= 40

cost of projection = 20

intermediate result ~ 20 pages

cost of selection = 0

intermediate result ~ 2000 pages

cost of SMJ = 3 ∗ (1000 + 50)

we pipeline the result
after the join operator

π_{ename}

$\sigma_{\text{dname} = \text{'Toy'}}$

Sort Merge Join

EMP DEPT

total I/O cost =
OUT
+20
+20 {materialize}
+0
+0 {pipeline}
+3150
= 3550 + OUT

EXAMPLE: COST ESTIMATION

buffer size B= 40

total I/O cost =
OUT
+20
+20 {materialize}
+1001
+1 {materialize}
+1
= 1043 + OUT

cost of projection = 20

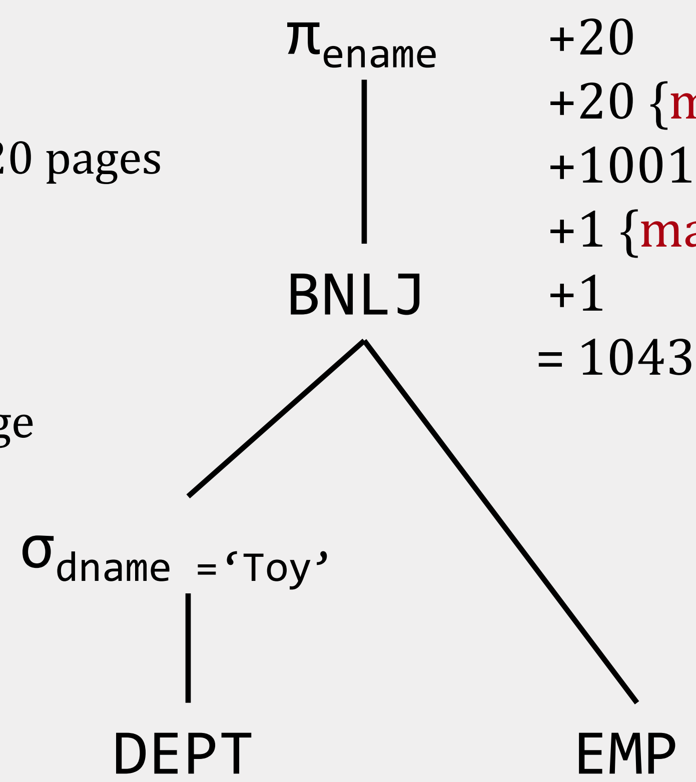
intermediate result ~ 20 pages

cost of BNLJ = 1000 + 1

intermediate result ~ 1 page

cost of selection ~ 1

use index on dname



Transaction Management

CS 564- Spring 2020

ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

- Transaction (TXN) management
- **ACID** properties
  - atomicity
  - consistency
  - isolation
  - durability
- Logging
- Scheduling & locking

TRANSACTIONS

DBMS MEMORY MODEL

Local: each process in a DBMS has its own local memory, where it stores values that only it "sees"

Global: each process can read from / write to shared data in main memory

Disk: global memory can read from / flush to disk

Log: *Assume on stable disk storage- spans both main memory and disk*

TRANSACTION

A **transaction** is a collection of *operations* that form a single *atomic* logical unit

```
BEGIN TRANSACTION ;
  {SQL}
COMMIT ;
```

- Operations: `READ / WRITE`
- In the real world, a TXN either happens completely or not at all

TRANSACTION EXAMPLES

- Bank transfer of money between two accounts
- Purchase a group of products online
- Register for a class (either waitlist or allocated)

TRANSACTIONS IN SQL

In SQL, multiple statements can be grouped together as a transaction:

```
BEGIN TRANSACTION ;
  UPDATE account
    SET balance = balance – 1000
    WHERE account_no = 1;
  UPDATE account
    SET balance = balance + 1000
    WHERE account_no = 2;
COMMIT ;
```

WHY TRANSACTIONS?

Grouping user actions (reads/writes) into *transactions* helps with two goals:

Recovery & Durability: keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

Concurrency: achieving better performance by parallelizing TXNs *without* inconsistencies

RECOVERY & DURABILITY

- Data must be durable in the face of:
  - system crashes
  - TXN aborts by the user

IDEA:

- make sure that TXNs are either *durably stored in full, or not at all*
- keep *log* to be able to *roll-back* TXNs

RECOVERY & DURABILITY: EXAMPLE

What can happen if the system crashes after the first SQL query is executed?

```
UPDATE account
  SET balance = balance – 1000
  WHERE account_no = 1;
UPDATE account
  SET balance = balance + 1000
  WHERE account_no = 2;
```

CONCURRENCY

Concurrent execution of user programs is essential for good DBMS performance

- better utilization: CPU/IO overlap
- avoids the situation where long running queries starve other queries
- provides the users with an illusion of a single-user system, called **isolation**
- maintains **consistency** during the concurrent execution

CONCURRENCY: EXAMPLE

What can happen if the two SQL queries are executed at the same time?

```
1: UPDATE account
   SET balance = balance – 1000
   WHERE account_no = 1;
2: UPDATE account
   SET balance = balance * 1.5
   WHERE account_no = 1;
```

THE ACID PROPERTIES

ACID PROPERTIES

Atomicity: all actions in the TXN happen, or none happen

Consistency: a database in a consistent state will remain in a consistent state after the TXN

Isolation: the execution of one TXN is isolated from other (possibly interleaved) TXNs

Durability: once a TXN commits, its effects must persist

ACID: ATOMICITY

Atomicity: All actions in the transaction happen, or none happen

- Two possible outcomes for a TXN
  - **commit**: all the changes are made
  - **abort**: no changes are made

ACID: CONSISTENCY

Consistency: a database in a consistent state will remain in a consistent state after the transaction

- Examples:
  - account number is unique
  - stock amount can't be negative
- How consistency is achieved:
  - the *programmer* makes sure a TXN takes a consistent state to a consistent state
  - the *DBMS* makes sure that the TXN is **atomic**

ACID: ISOLATION

Isolation: the execution of one transaction is isolated from other (possibly interleaved) transactions

Example:

- if T1, T2 are interleaved, the result should be the same as executing first T1 then T2, or first T2 then T1

ACID: DURABILITY

Durability: if a transaction commits, its effects must persist

- for example, if the system crashes after a commit, the effects must remain
- essentially, this means that we have to write to disk

CHALLENGES FOR ACID

- in spite of failures: power failures, but not media failures
- users may abort the program: need to “rollback the changes”
  - we need to *log* what happened!
- many users can execute concurrently
  - *locking* (we’ll see this next lecture!)

all these must be done while keeping performance in mind!

LOGGING

WHY LOGGING?

- Can we just write the modified pages to disk **only** once whole TXN is complete?
  - if abort/crash and the TXN is not complete, it has no effect: atomicity + durability!

- However, we need to **log** partial results of TXNs:
  - memory constraints (the buffer pool may want to write pages to disk earlier!)
  - time constraints (what if one TXN takes very long?)

LOGGING

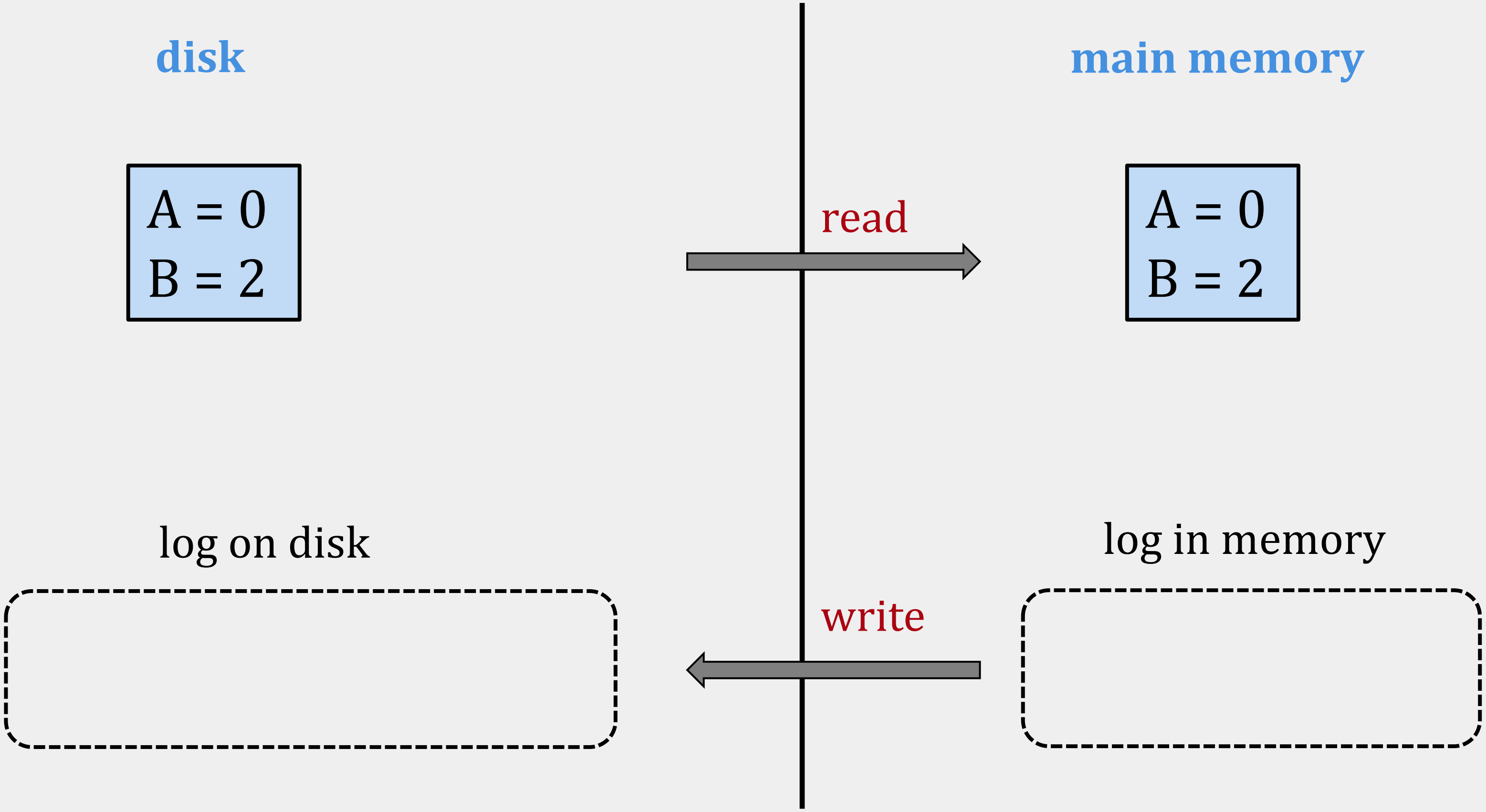
The **log** is a list of modifications

- it records **REDO/UNDO** information for every update
  - only minimal info (diff) written to log
- it is *duplexed* and *archived* on stable storage (disk)
- it can **force** pages to disk

- it consists of an *ordered list* of actions of the form

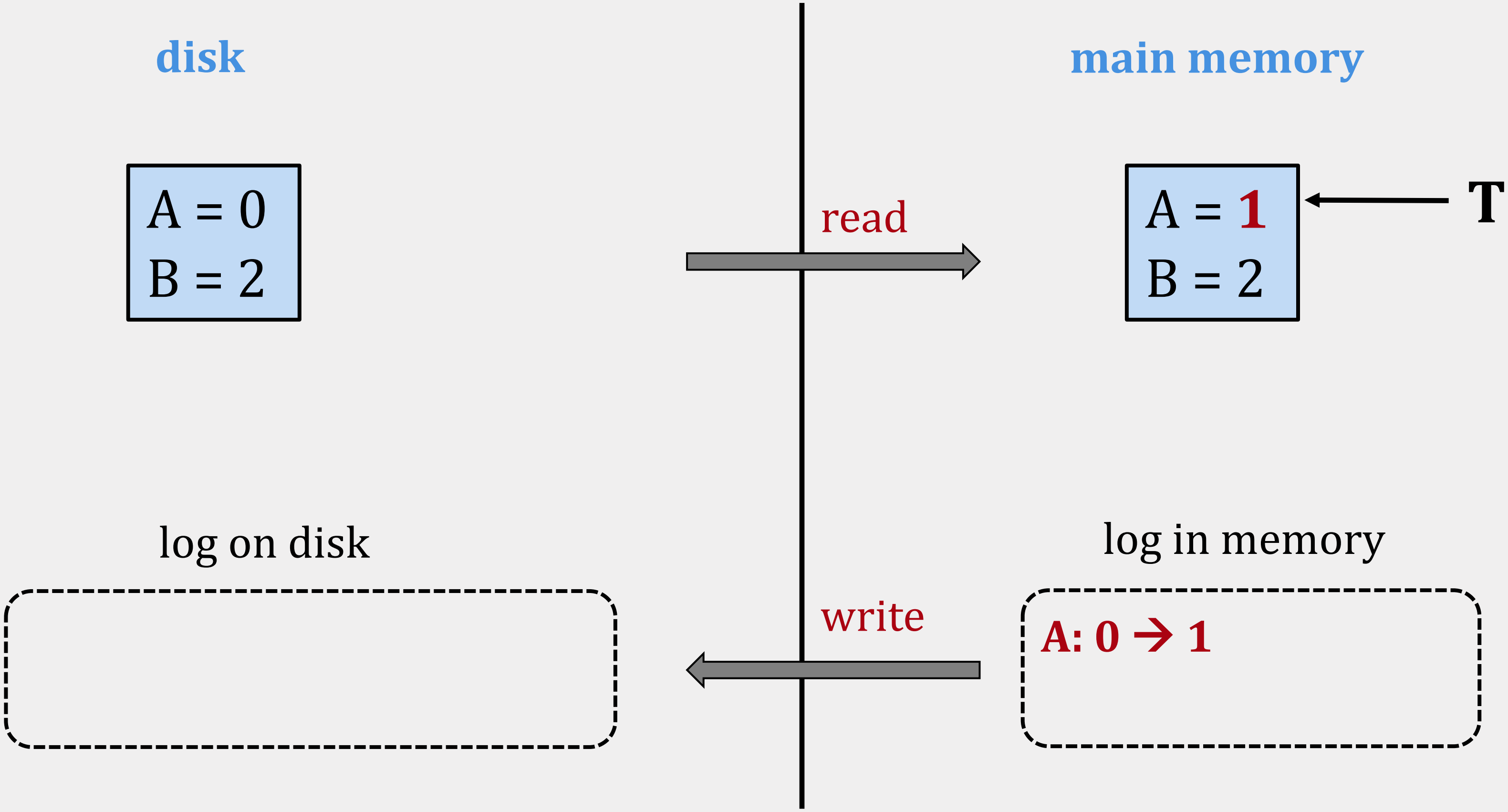
  `<TXNID, location, old-data, new-data>`

LOGGING: EXAMPLE



T: Read(A), Write(A)

LOGGING: EXAMPLE



The log records the operation in the main memory!

T: Read(A), Write(A)

HOW DO WE WRITE THIS TO DISK?

- We will see the Write-Ahead Logging (WAL) protocol
- WAL guarantees atomicity & durability
- We will also see why other ideas don't work!

WRITE-AHEAD LOGGING

1. we force the log record for an update to disk before the corresponding page goes to disk

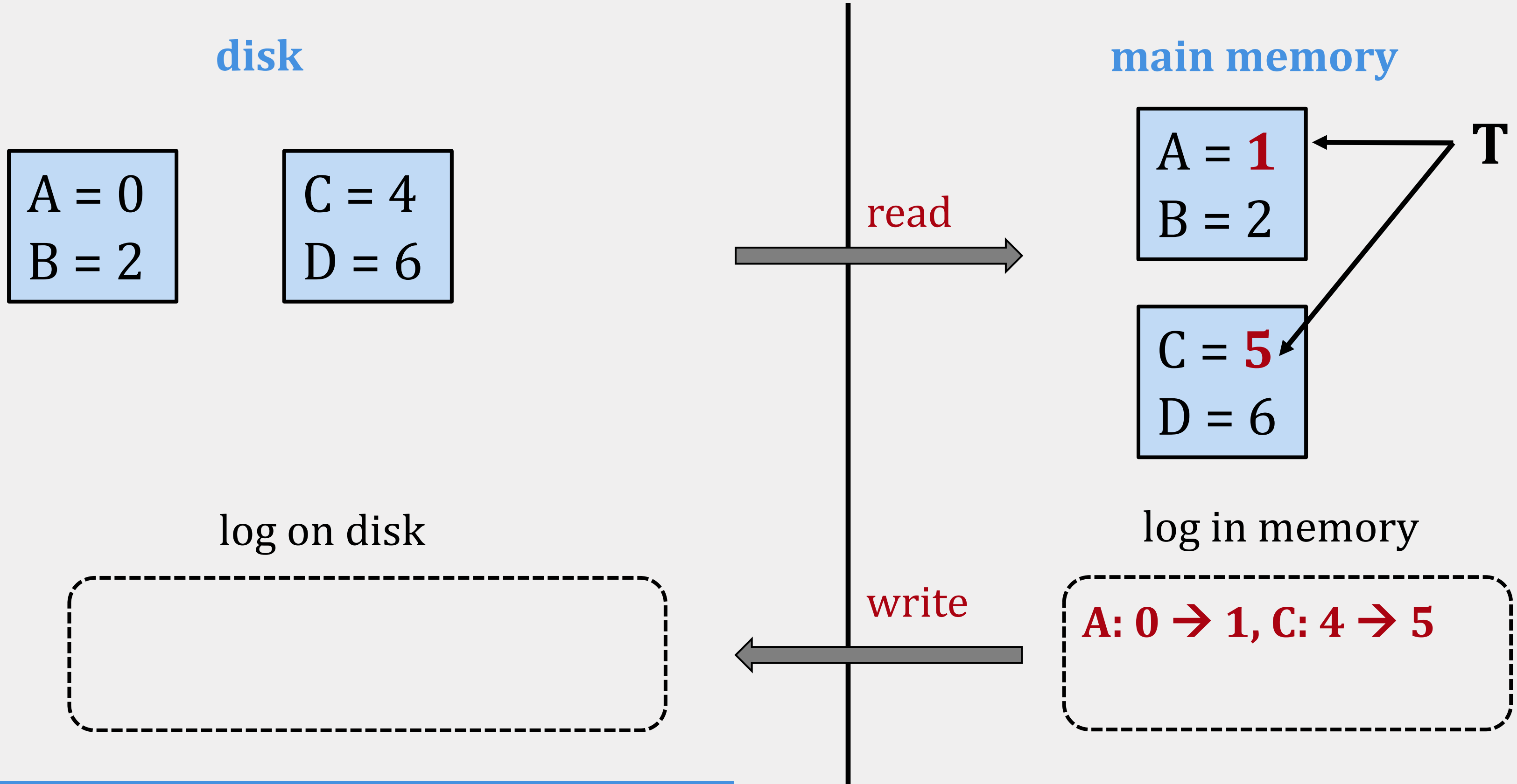
ATOMICITY

2. we write to disk all log records for a TXN before commit

DURABILITY

Note: WAL does not record any reads, only updates!

LOGGING: BAD PROTOCOLS #1

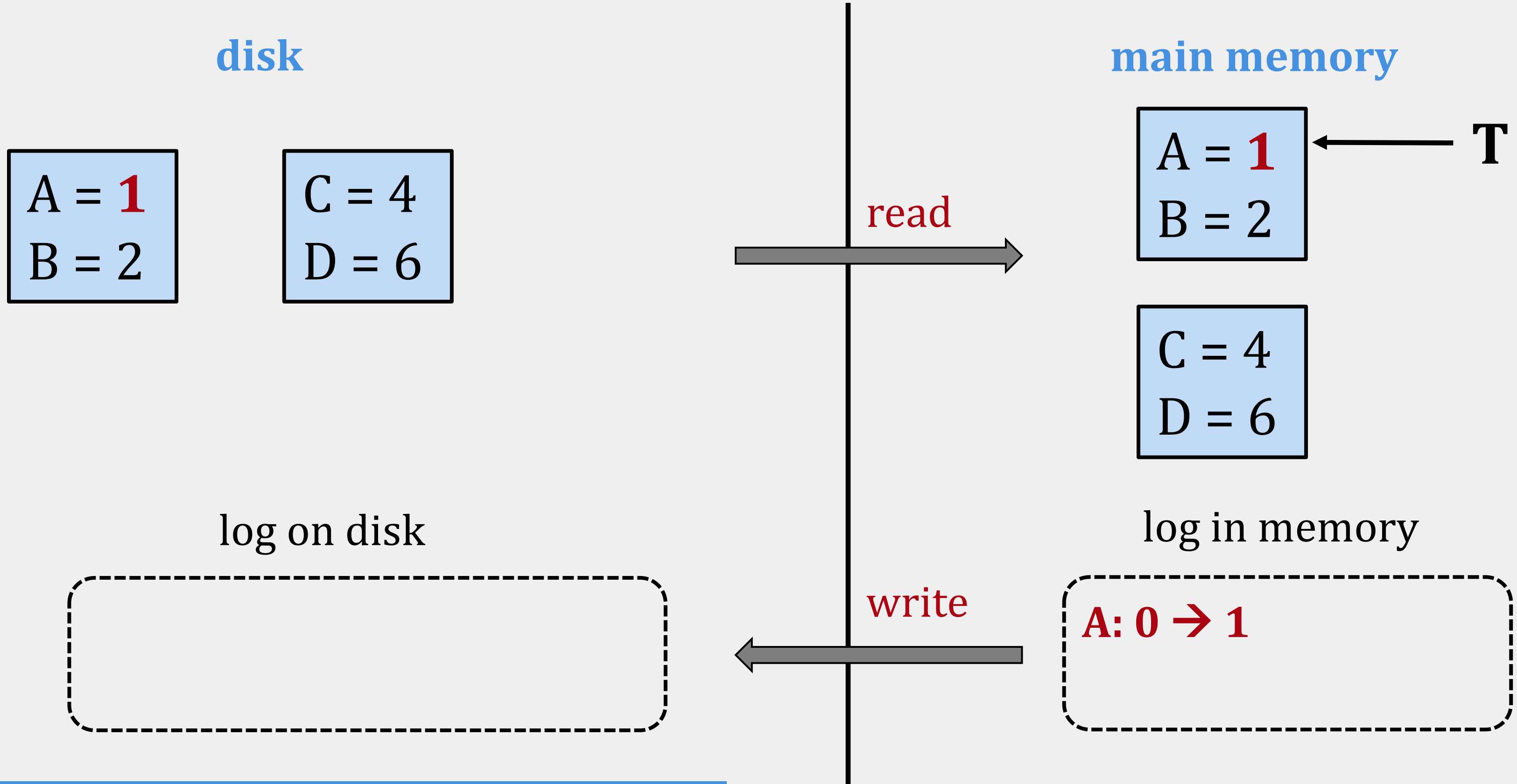


What happens if we commit the TXN before writing page/log to disk?

- if crash, not durable!

T: Write(A), Write(C)

LOGGING: BAD PROTOCOLS #2

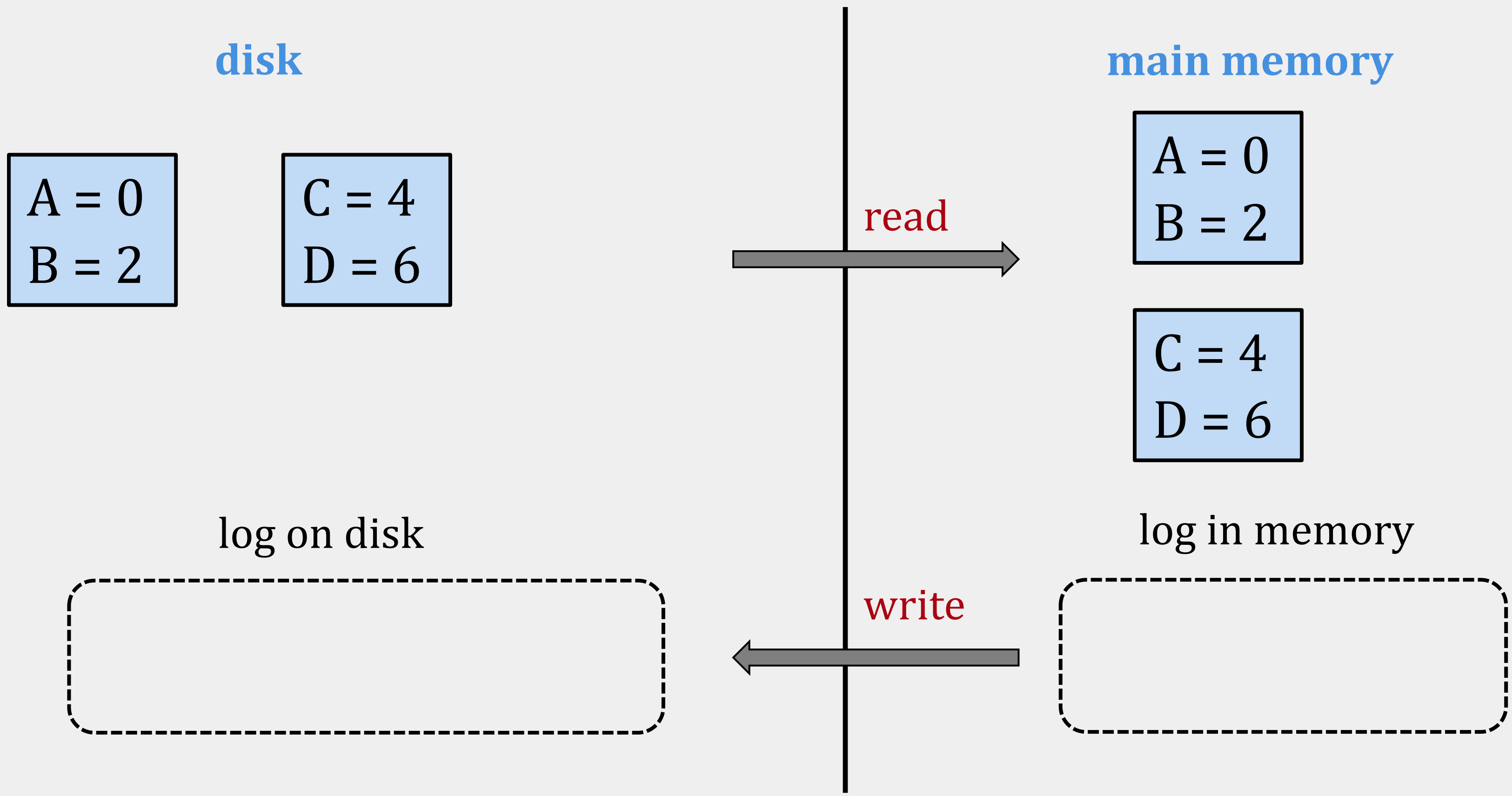


What happens if we write the page to disk before the log writes to disk?

- if crash/abort, not atomicity!

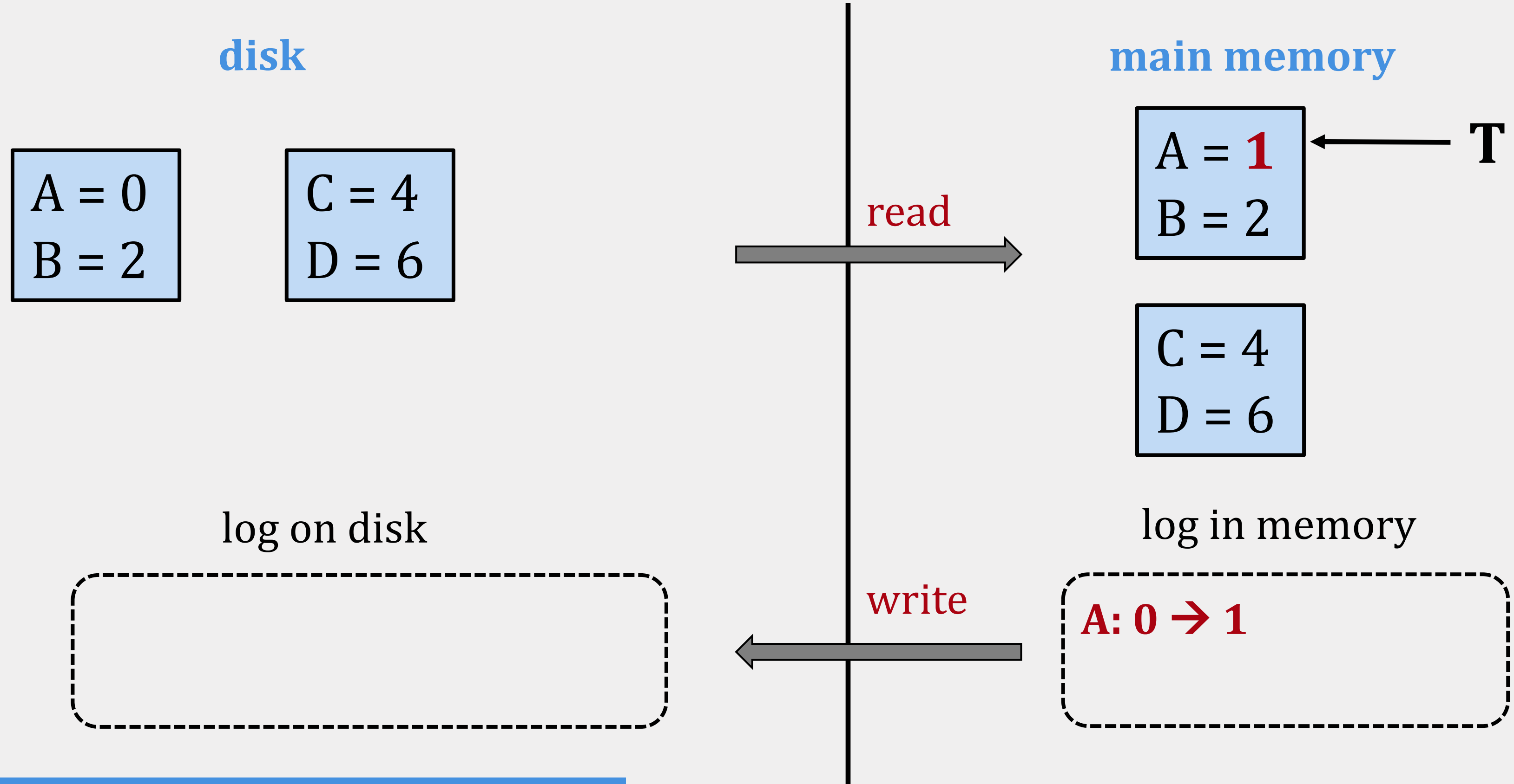
T: Write(A), Write(C)

LOGGING: WAL PROTOCOL



T: Write(A), Write(C)

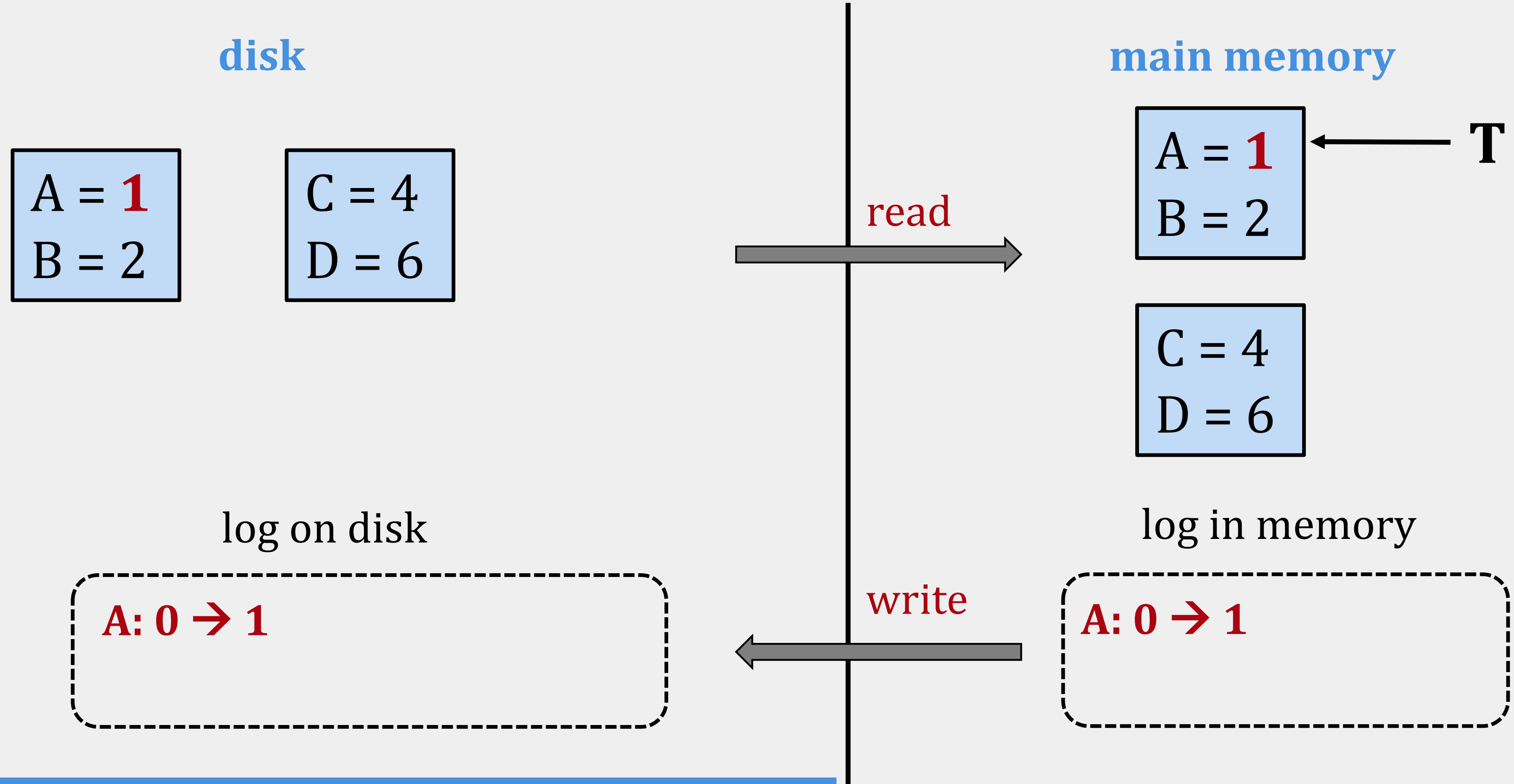
LOGGING: WAL PROTOCOL



So far no writing to disk.
If crash/abort now, we are fine!

T: Write(A), Write(C)

LOGGING: WAL PROTOCOL

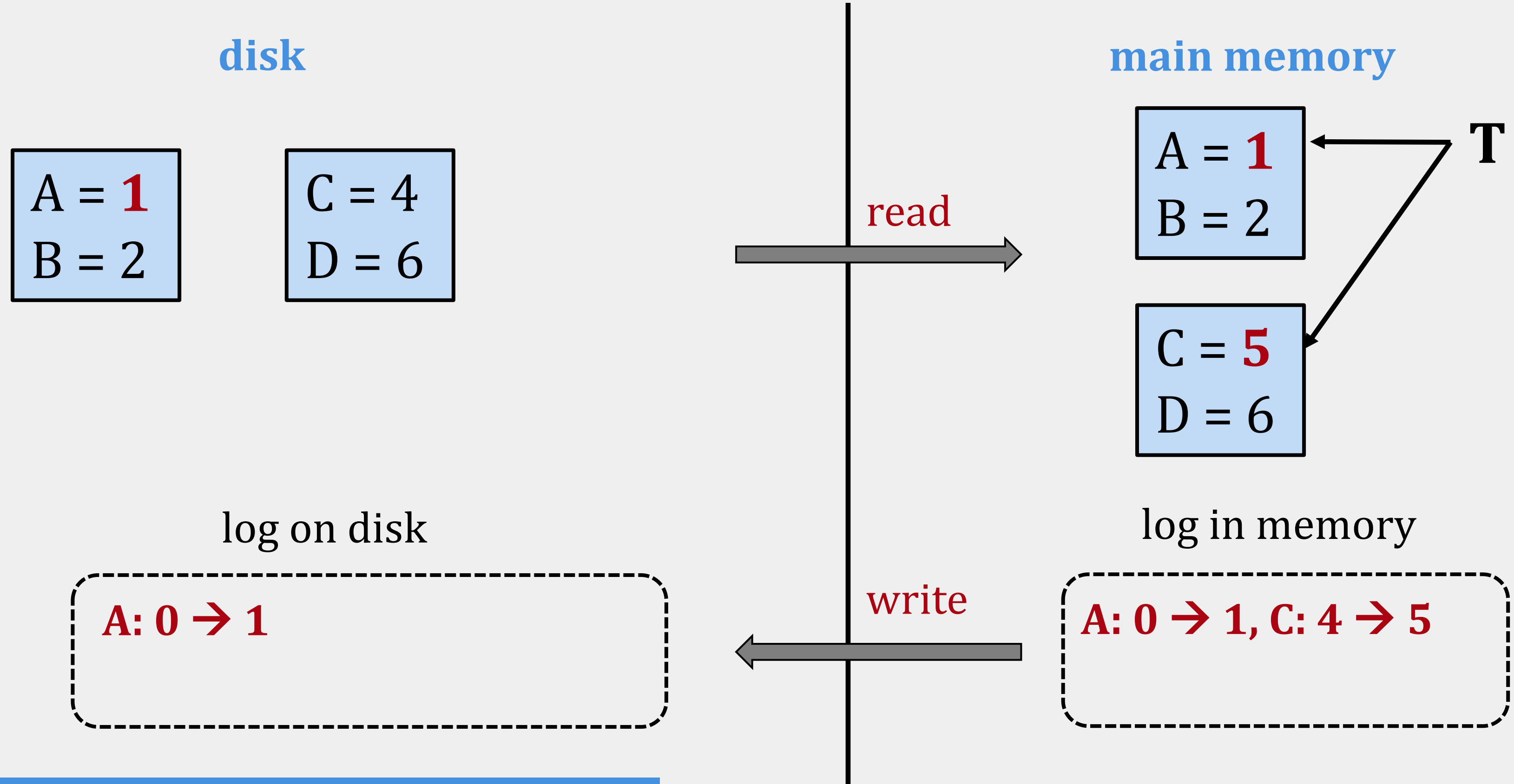


If the buffer decides to write to disk, we must write the log before!

- if crash/abort, we can UNDO using log

T: Write(A), Write(C)

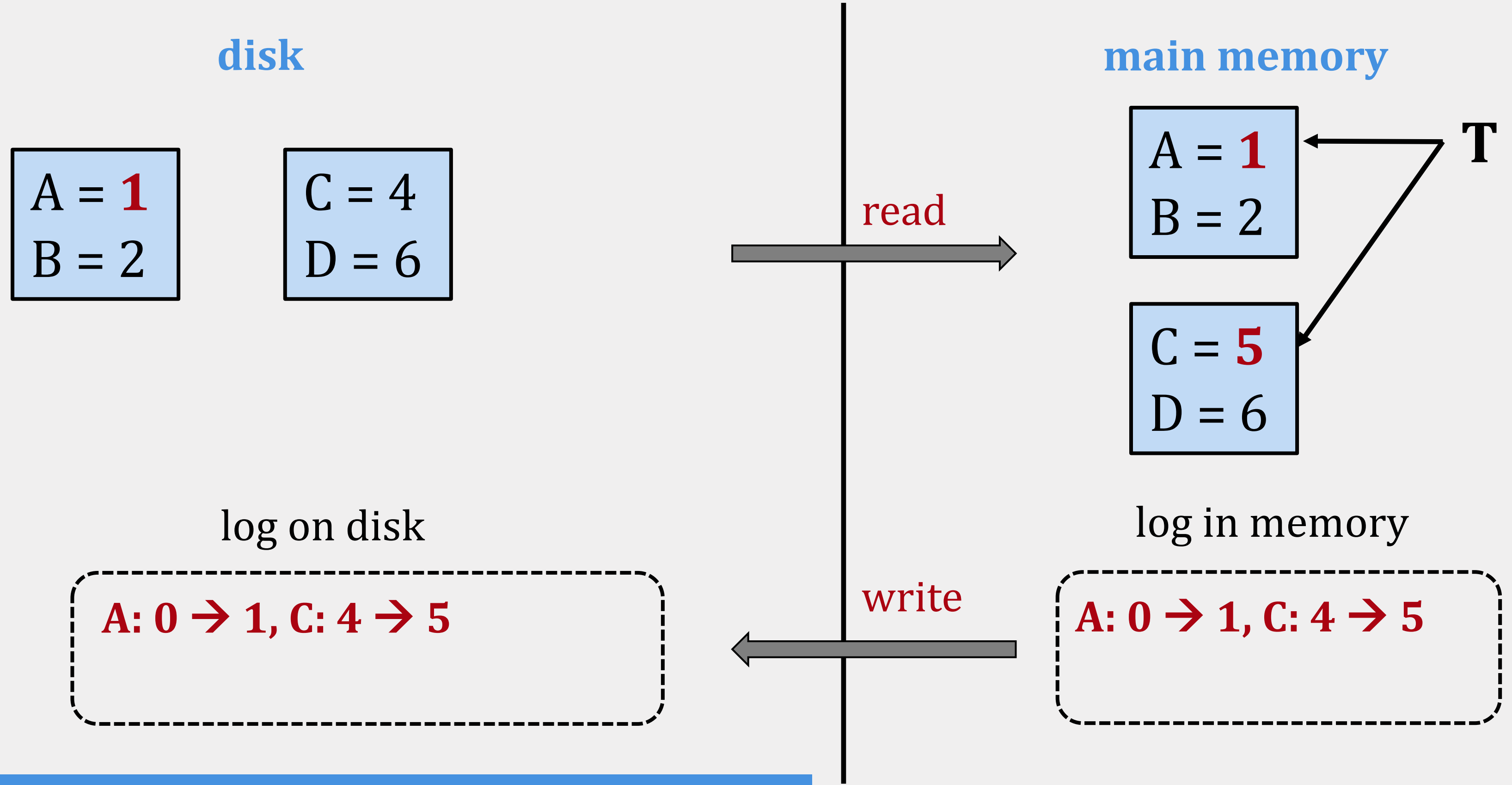
LOGGING: WAL PROTOCOL



We don’t write to disk right away

T: Write(A), Write(C)

LOGGING: WAL PROTOCOL

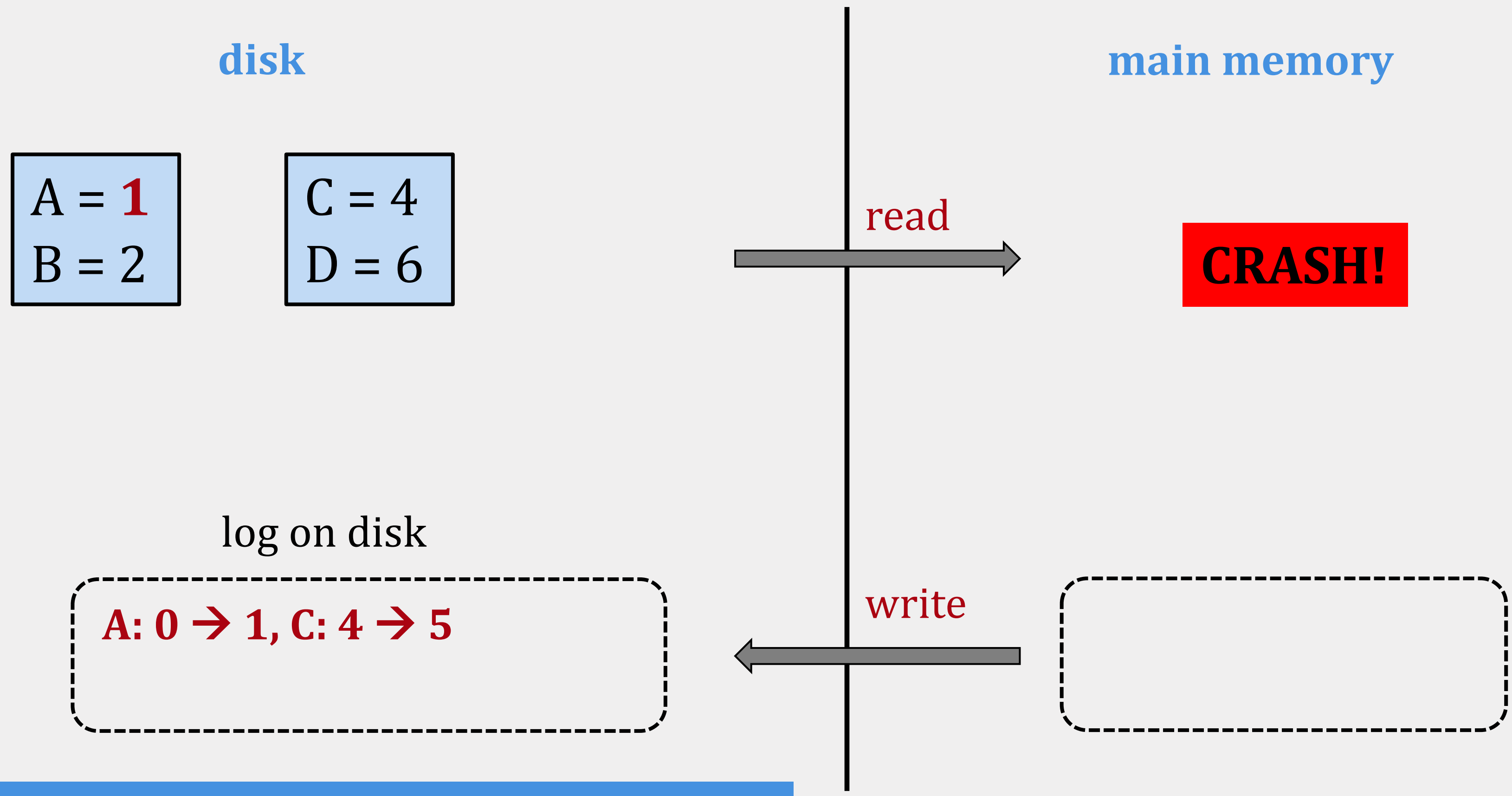


Upon commit of the TXN, we force write the log to disk!

- the page does not need to be written!

T: Write(A), Write(C)

LOGGING: WAL PROTOCOL



Upon crash, we can use the log to reconstruct (REDO) the modified pages

T: Write(A), Write(C)

ARIES

- The WAL protocol still has to force multiple pages to disk, which can limit performance
- ARIES is a (very) complex recovery algorithm that improves performance and has 3 phases:
  – Analysis
  – UNDO (rollback)
  – REDO (replay)

- For more on crashes and recovery, take CS 764!

TRANSACTION MANAGEMENT II

CS 564- Spring 2020

ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

- Transaction (TXN) management
- **ACID** properties
  - atomicity
  - consistency
  - isolation
  - durability
- Logging
- Scheduling & locking

ACID PROPERTIES: RECAP

Atomicity: all actions in the TXN happen, or none happen

Consistency: a database in a consistent state will remain in a consistent state after the TXN

Isolation: the execution of one TXN is isolated from other (possibly interleaved) TXNs

Durability: once a TXN commits, its effects must persist

CONCURRENCY

CONCURRENCY

- The DBMS runs multiple TXNs concurrently
- To achieve better performance, interleaving the operations of the TXNs is critical
  - possibly slow TXNs
  - CPU/IO overlap
- But interleaving can lead to problems!

Remember: we must guarantee **isolation** & **consistency**!

EXAMPLE

T1: *transfer $100 from A to B*

```
BEGIN TRANSACTION ;
  UPDATE account
    SET balance = balance – 100
    WHERE account_name = A;
  UPDATE account
    SET balance = balance + 100
    WHERE account_name = B;
COMMIT ;
```

T2: *add 10% interest to both accounts*

```
BEGIN TRANSACTION ;
  UPDATE account
    SET balance = balance * 1.1
COMMIT ;
```

Let’s see how the DBMS can schedule the 2 transactions

EXAMPLE

First run T1, then run T2

T1	***T2***
A ← A - 100	
B ← B + 100	
	A ← A * 1.1
	B ← B * 1.1

time

Beginning

- A = 200, B = 100

End

- A = 110, B = 220

This is called a **<u>serial</u>** schedule

EXAMPLE

First run T2, then run T1

T1	*T2*
	A ← A * 1.1
	B ← B * 1.1
A ← A - 100	
B ← B + 100	

time

Beginning
- A = 200, B = 100

End
- A = 120, B = 210

This is also a serial schedule

EXAMPLE

Interleaving the operations of T1 and T2

T1	*T2*
	A ← A * 1.1
A ← A - 100	
	B ← B * 1.1
B ← B + 100	

time

Beginning
- A = 200, B = 100

End
- A = 120, B = 210

Same result as if we run serially T2 and then T1! This is called a **serializable** schedule

EXAMPLE

Different interleaving of the operations of T1 and T2

T1	*T2*
	A ← A * 1.1
A ← A - 100	
B ← B + 100	
	B ← B * 1.1

time

Beginning
- A = 200, B = 100

End
- A = 120, B = 220

Different result from both serial schedules!
This is called a **not serializable** schedule

SCHEDULES: DEFINITIONS

Schedule: an interleaving of actions from a set of TXNs, where the actions of any TXN are in the original order

Serial schedule: a schedule where there is no interleaving of actions from different TXNs

Equivalent schedules: two schedules are equivalent if *for every* database state, they will have the same effect

Serializable schedule: a schedule that is equivalent to *some* serial schedule

Note: we assume that all TXNs commit in the schedules!

THE DBMS'S VIEW OF THE SCHEDULE

T1	T2
	A ← A * 1.1
A ← A - 100	
B ← B + 100	
	B ← B * 1.1

time

Each action is a read (**R**) followed by a write (**W**)

T1	T2
	R(A)
	W(A)
R(A)	
W(A)	
R(B)	
W(B)	
	R(B)
	W(B)

CONFLICTS IN SCHEDULES

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

- Write-Read conflict
- Read-Write conflict
- Write-Write conflict

A conflict does not always lead to a problem when interleaving!

CONFLICTS VS ANOMALIES

Conflicts help us characterize different schedules

- present in both "good" and "bad" schedules

Anomalies are instances where isolation and/or consistency is broken because of a "bad" schedule

- we often characterize different anomaly types by what types of conflicts predicated them

DIRTY READ

T1	*T2*
	W(A)
R(B)	
R(A)	
Commit	
	W(C)

time

A **dirty read** occurs when a TXN reads data that was modified by a not yet committed TXN

- in the example, T1 reads A, which was previously modified by T2
- occurs because of a W-R conflict!

If T2 aborts, this will lead to inconsistency!

UNREPEATABLE READ

T1	*T2*
	R(A)
W(A)	
R(B)	
Commit	
	R(A)

time

An **unrepeatable read** occurs when a TXN reads data twice, but in between the data was modified by another TXN

- in the example, T2 reads A, T1 then modifies T1, and T2 reads again
- occurs because of a R-W conflict!

OVERWRITING UNCOMMITTED DATA

T1	*T2*
	W(A)
W(A)	
W(B)	
Commit	
	W(B)

time

This occurs when a TXN overwrites the data of an uncommitted TXN

- in the example, the last version of A and B would not be consistent with any serial schedule
- occurs because of a W-W conflict!

Conflict Serializability

CONFLICT SERIALIZABILITY

- Two schedules are **conflict equivalent** if:
  - they involve *the same actions of the same TXNs*
  - every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- A schedule is **conflict serializable** if it is *conflict equivalent* to *some* serial schedule
- This provides us with a way to distinguish "good" from "bad" schedules

Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation

EXAMPLE

T1	T2
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	
R(B)	
W(B)	

- In both, W(A) in T2 comes before R(A) in T1
- The same happens with all other pairs of conflicting actions
- Since the left schedule is serial, the right schedule is conflict serializable!

T1	T2
	R(A)
	W(A)
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	

EXAMPLE

T1	T2
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	
R(B)	
W(B)	

- The order has changed now!
- The two schedules are not conflict equivalent
- We still need to check all other serial schedules!

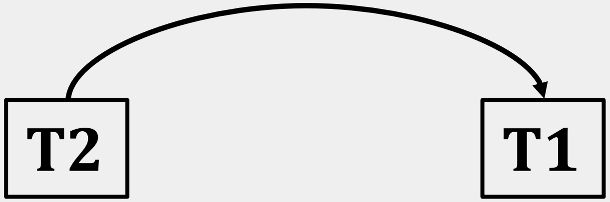
T1	T2
	R(A)
R(A)	
W(A)	
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

THE CONFLICT GRAPH

- The conflict graph looks at conflicts at the transaction level
- the nodes are TXNs
- there is an edge from T_i to T_j *if any actions in T_i precede and conflict with any actions in T_j*

THE CONFLICT GRAPH

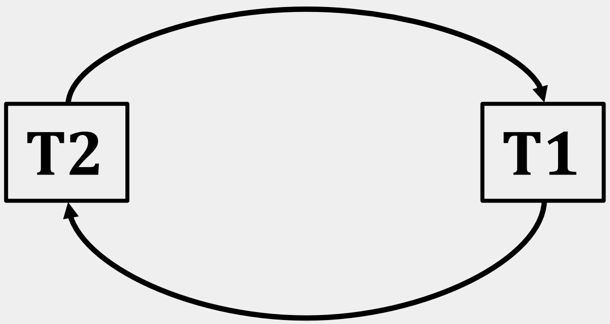
T1	*T2*
	R(A)
	W(A)
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	



- Since W(A) in T2 is before R(A) in T1, we add an edge from T2 to T1
- There is no edge from T1 to T2 in this case!

THE CONFLICT GRAPH

T1	*T2*
	R(A)
R(A)	
W(A)	
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

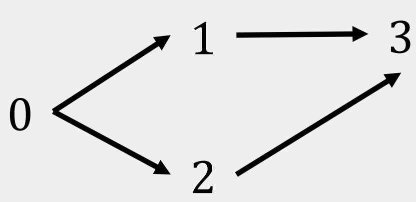


- Since R(A) in T1 is before W(A) in T2, we add an edge from T1 to T2
- Since W(B) in T2 is before R(B) in T1, we also add an edge from T2 to T1

THE CONFLICT GRAPH: THEOREM

> **Theorem**: a schedule is conflict serializable if and only if its conflict graph is acyclic (i.e. it has no directed cycles)

- A topological ordering of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A directed acyclic graph (DAG) always has one or more topological orderings
  – if there are cycles, there exists no such ordering!

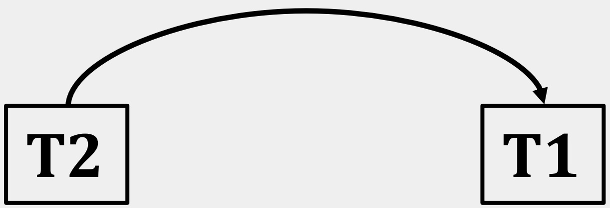


There are 2 possible topological orderings:

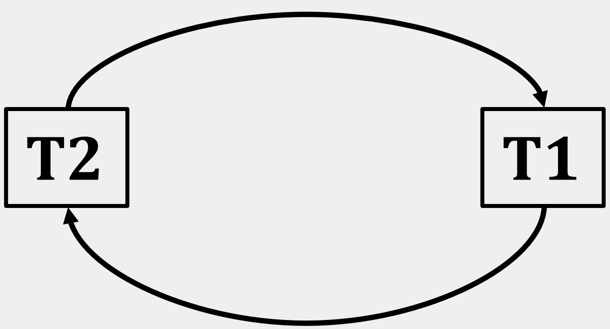
- 0, 2, 1, 3
- 0, 1, 2, 3

THE CONFLICT GRAPH

- In the conflict graph, a topological ordering of the nodes corresponds to a serial ordering of TXNs (serial schedule)
- Thus an **acyclic** conflict graph → conflict serializable!



top ordering: T2, T1
this is conflict equivalent to a serial schedule with first T2, then T1



there is a cycle, so no topological ordering
not conflict serializable!

LOCKING

LOCKING

- Locking is a technique for concurrency control
- Lock information maintained by a *lock manager*:
  - stores (TID, RID, Mode) triples
  - mode is either Shared (S) or Exclusive (X)

	--	S	X
--	√	√	√
S	√	√	
X	√		

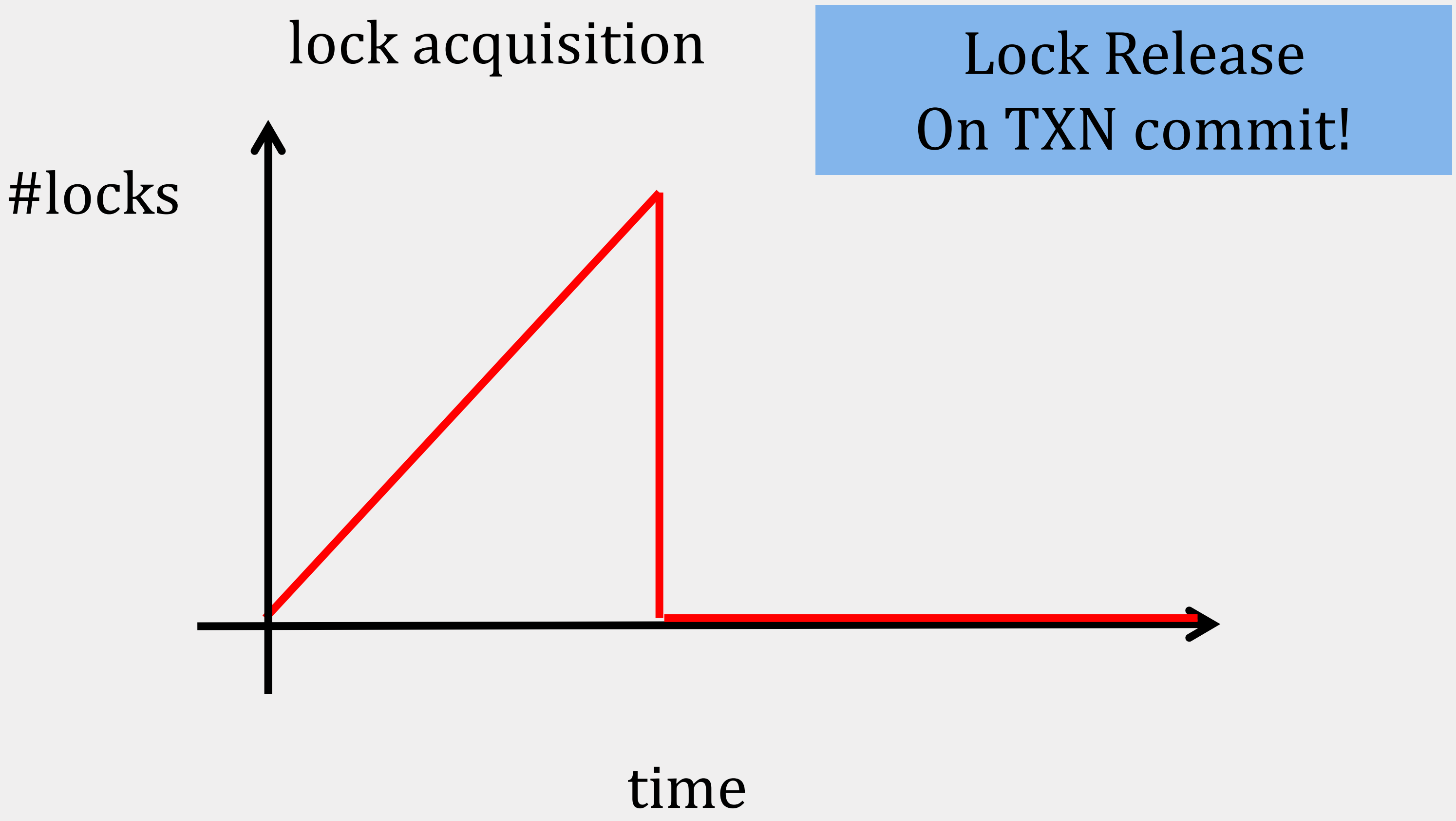
- If a transaction cannot get a lock, it has to wait in a queue

STRICT 2 PHASE LOCKING

- Each transaction must obtain a **S** lock on object before reading, and an **X** lock on object before writing
- If a transaction holds an **X** lock on an object, no other transaction can get a lock (S or X) on that object
- All locks held by a transaction are released only when the transaction completes

Strict 2PL guarantees conflict serializability!

STRICT 2PL: FIGURE



STRICT 2PL

- If a schedule follows strict 2PL and locking, it is conflict serializable
  - and thus serializable
  - and thus maintains isolation & consistency!

- Not all serializable schedules are allowed by strict 2PL
- But running a strict 2PL protocol has some issues!

DEADLOCKS

T1	*T2*
R(B)	
W(B)	
	R(A)
	W(A)
R(A)	
	R(B)

T1 gets an X-lock on B

T2 gets an X-lock on A

T1 wants to read A, but has to wait…

T2 wants to read B, but also has to wait…

We now have a **deadlock**!

DEADLOCKS

- Deadlocks can cause the system to wait forever
- We need to detect deadlocks and break, or prevent deadlocks
- Simple mechanism: timeout and abort
- More sophisticated methods exist

PERFORMANCE OF LOCKING

- Locks have a performance penalty:
  - blocked actions
  - aborted transactions
- Because of blocking, we can not increase forever the throughput of transactions
- At the point where the throughput cannot increase, we say that the system thrashes

Transactions in SQL

TRANSACTIONS IN SQL

- What object should we lock?

```
SELECT COUNT(*)
FROM   Employee
WHERE  age = 20 ;
```

- We can apply locking at different granularities:
  - lock the whole table Employee
  - lock only the rows with age = 20

TRANSACTIONS IN SQL

Transaction characteristics:

- Access mode: `READ ONLY, READ WRITE`
- Isolation level
  - Serializable: default (Strict 2PL)
  - Repeatable reads: (R/W locks, but phantom can occur)
    - Read only committed records
    - Between two reads by the same transaction, no updates by another transaction
  - Read committed (W locks longterm, R locks shortterm)
    - Read only committed records
  - Read uncommitted (only reads, no locks)