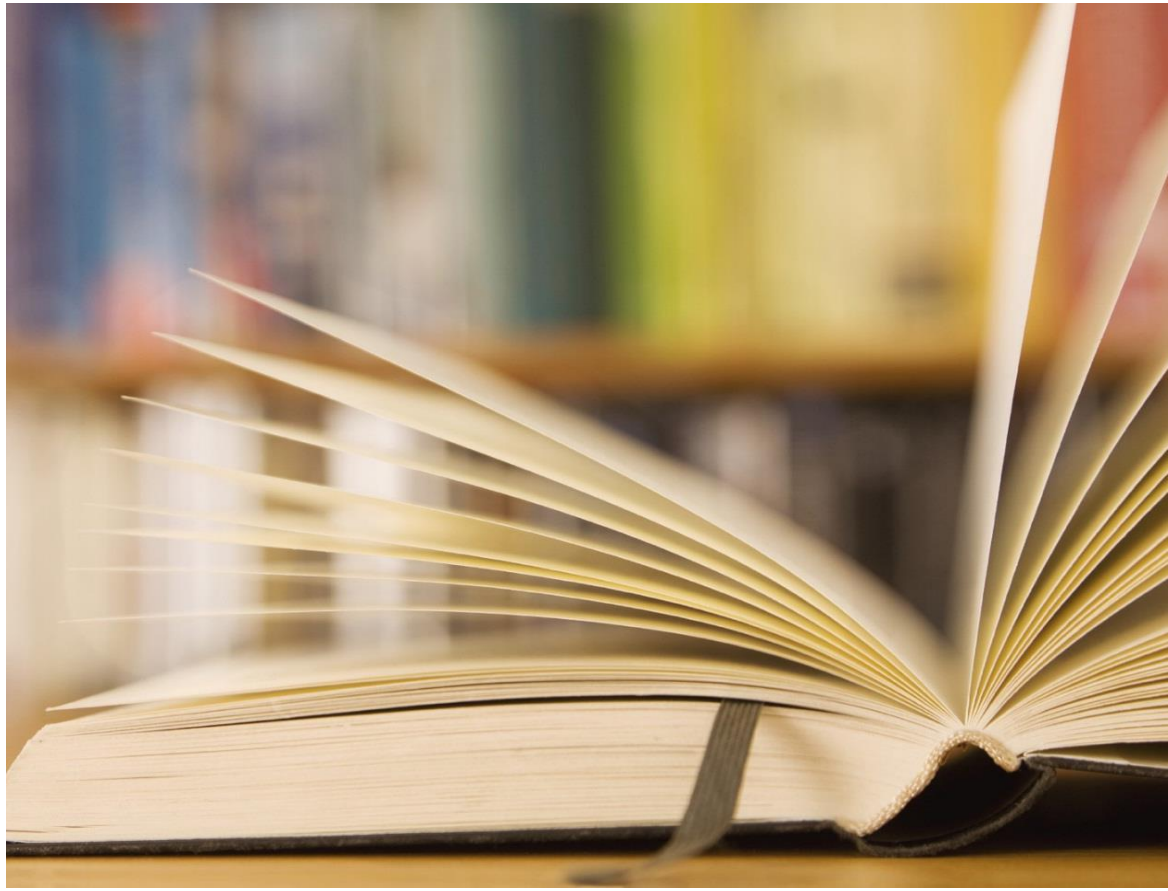
# CSCI317 – Database Performance Tuning

2019 S1 Final Examination

20 November 2019



**Introduction**

**The questions 2, 4, 5, 6, and 7 of the examination paper are related to the following simplified version of TPC-HR benchmark database used in the laboratory classes.**

```
CUSTOMER( C_CUSTKEY NUMBER(12) NOT NULL,
C_NAME VARCHAR(25) NOT NULL,
C_ADDRESS VARCHAR(40) NOT NULL,
C_NATIONKEY NUMBER(12) NOT NULL,
CONSTRAINT CUSTOMER_PKEY PRIMARY KEY(C_CUSTKEY) );
PART( P_PARTKEY NUMBER(12) NOT NULL,
P_NAME VARCHAR(55) NOT NULL,
P_BRAND CHAR(10) NOT NULL,
P_SIZE NUMBER(12) NOT NULL,
P_RETAILPRICE NUMBER(12,2) NOT NULL,
CONSTRAINT PART_PKEY PRIMARY KEY (P_PARTKEY) );
PARTSUPP( PS_PARTKEY NUMBER(12) NOT NULL,
PS_SUPPNAME VARCHAR(55) NOT NULL,
PS_AVAILQTY NUMBER(12) NOT NULL,
CONSTRAINT PARTSUPP_PKEY PRIMARY KEY (PS_PARTKEY,PS_SUPPNAME),
CONSTRAINT PARTSUPP_FKEY FOREIGN KEY(PS_PARTKEY)
REFERENCES PART(P_PARTKEY) );
```

```
ORDERS( O_ORDERKEY NUMBER(12) NOT NULL,
O_CUSTKEY NUMBER(12) NOT NULL,
O_TOTALPRICE NUMBER(12,2) NOT NULL,
O_ORDERDATE DATE NOT NULL,
CONSTRAINT ORDERS_PKEY PRIMARY KEY (O_ORDERKEY),
CONSTRAINT ORDERS_FKEY1 FOREIGN KEY (O_CUSTKEY)
```
REFERENCES CUSTOMER(C_CUSTKEY) );
```
LINEITEM( L_ORDERKEY NUMBER(12) NOT NULL,
L_PARTKEY NUMBER(12) NOT NULL,
L_LINENUMBER NUMBER(12) NOT NULL,
L_QUANTITY NUMBER(12,2) NOT NULL,
L_SHIPDATE DATE NOT NULL,
```
CONSTRAINT LINEITEM_PKEY PRIMARY KEY (L_ORDERKEY, L_LINENUMBER),
CONSTRAINT LINEITEM_FKEY1 FOREIGN KEY (L_ORDERKEY)
REFERENCES ORDERS(O_ORDERKEY),
CONSTRAINT LINEITEM_FKEY2 FOREIGN KEY (L_PARTKEY)
REFERENCES PART(P_PARTKEY) );

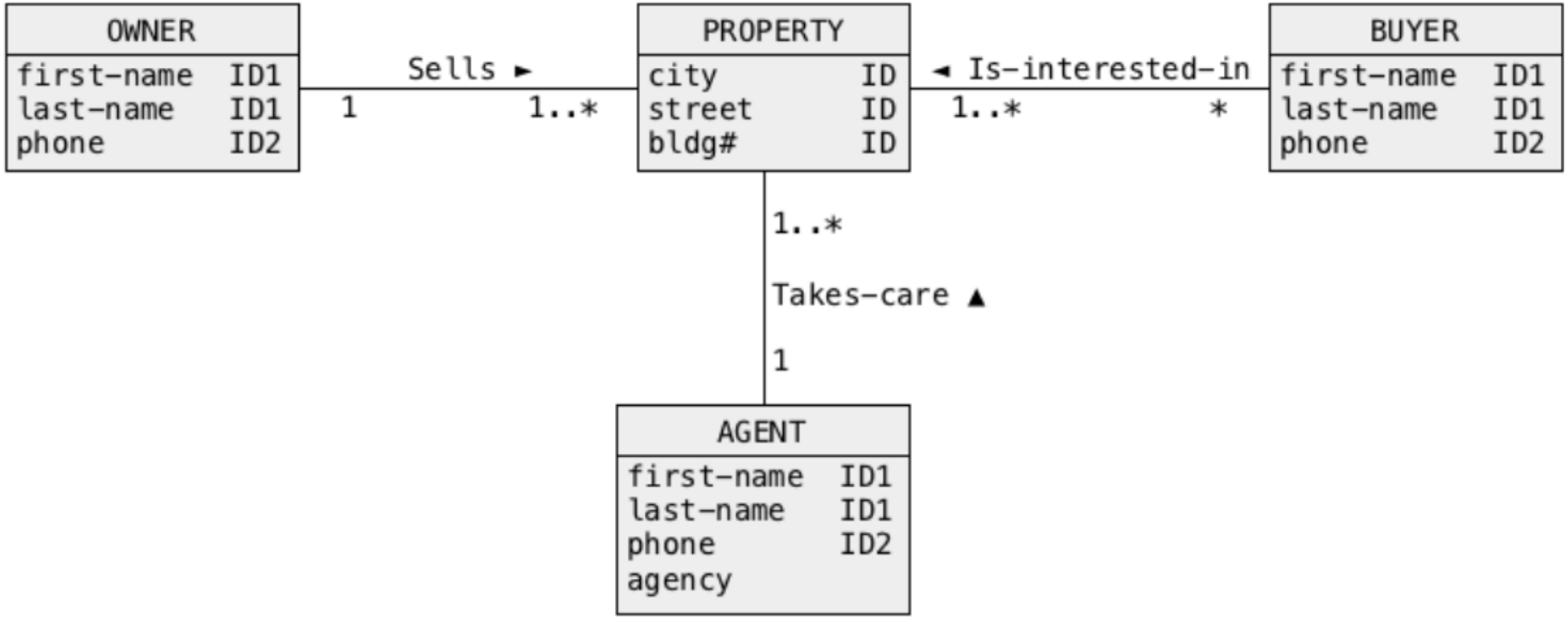
Assume that, the relational tables listed above occupy the following amounts of disk storage:

```
CUSTOMER 100 Mbytes
PART 30 Mbytes
PARTSUPP 400 Mbytes
ORDERS 500 Mbytes
LINEITEM 900 Mbytes
```

# 2019S1, Q1

The following conceptual schema represents a database domain where owners sell real estate properties, buyers are  interested in real estate properties and sellers take care about real estate properties.

# 2019S1, Q1


OWNER
first-name ID1
last-name ID1
phone ID2
Sells ►
1
1..*
PROPERTY
city ID
street ID
bldg# ID
◄ Is-interested-in
1..*
*
BUYER
first-name ID1
last-name ID1
phone ID2
1..*
Takes-care ▲
1
AGENT
first-name ID1
last-name ID1
phone ID2
agency

# 2019S1, Q1

An objective of this task is to use denormalization to improve the performance of the following class of applications.

*Find the phones of buyers (attribute phone in the class BUYER) who are interested in the real estate properties located in a given city (attribute city in the class PROPERTY) and being taken care about by an agent from a given agency (attribute agency in he class AGENT)*

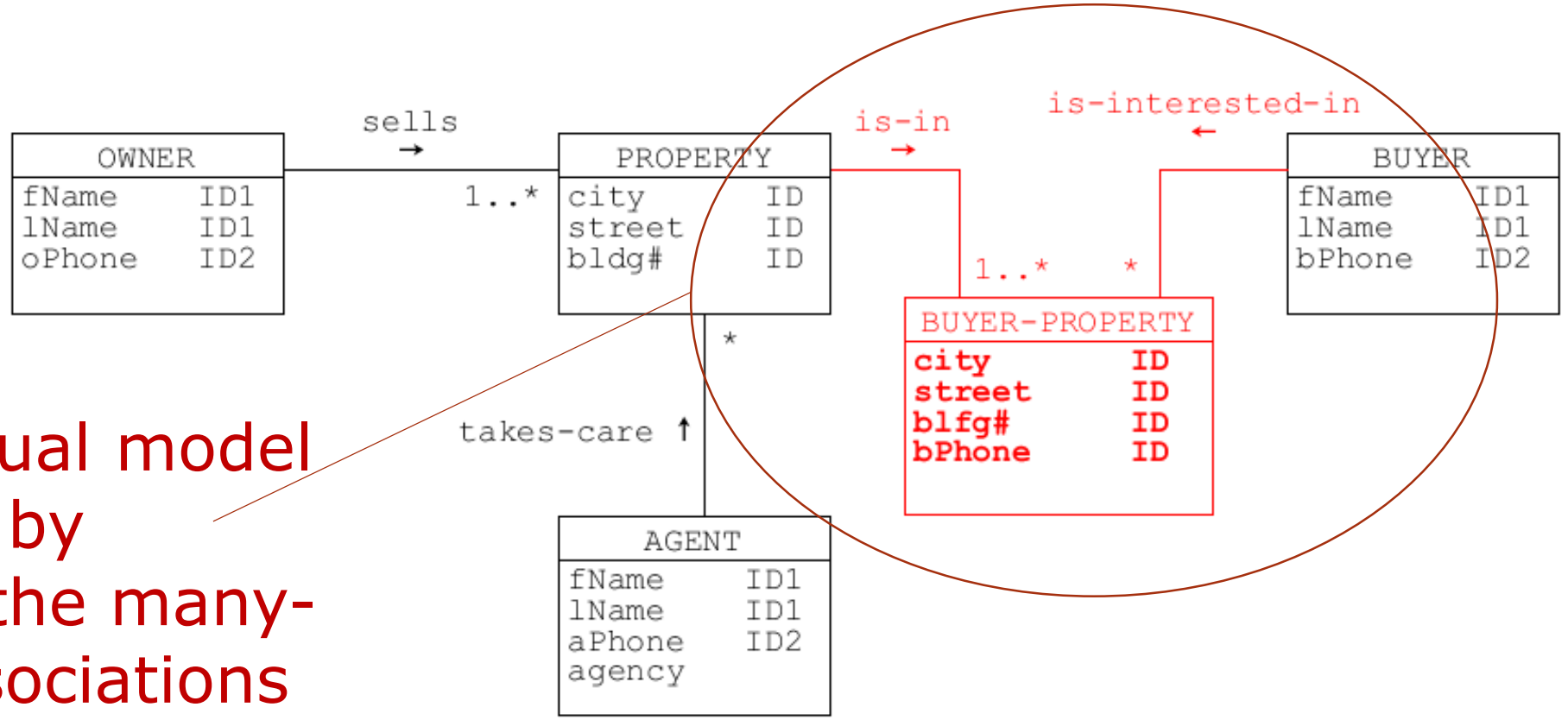
# 2019S1, Q1

A sample application that belongs to a class described above could be the following.

*Find the phones of buyers who are interested in the real estate property located in Sydney and being taken care about by an agent from an agency Real Estate Demolishers.*

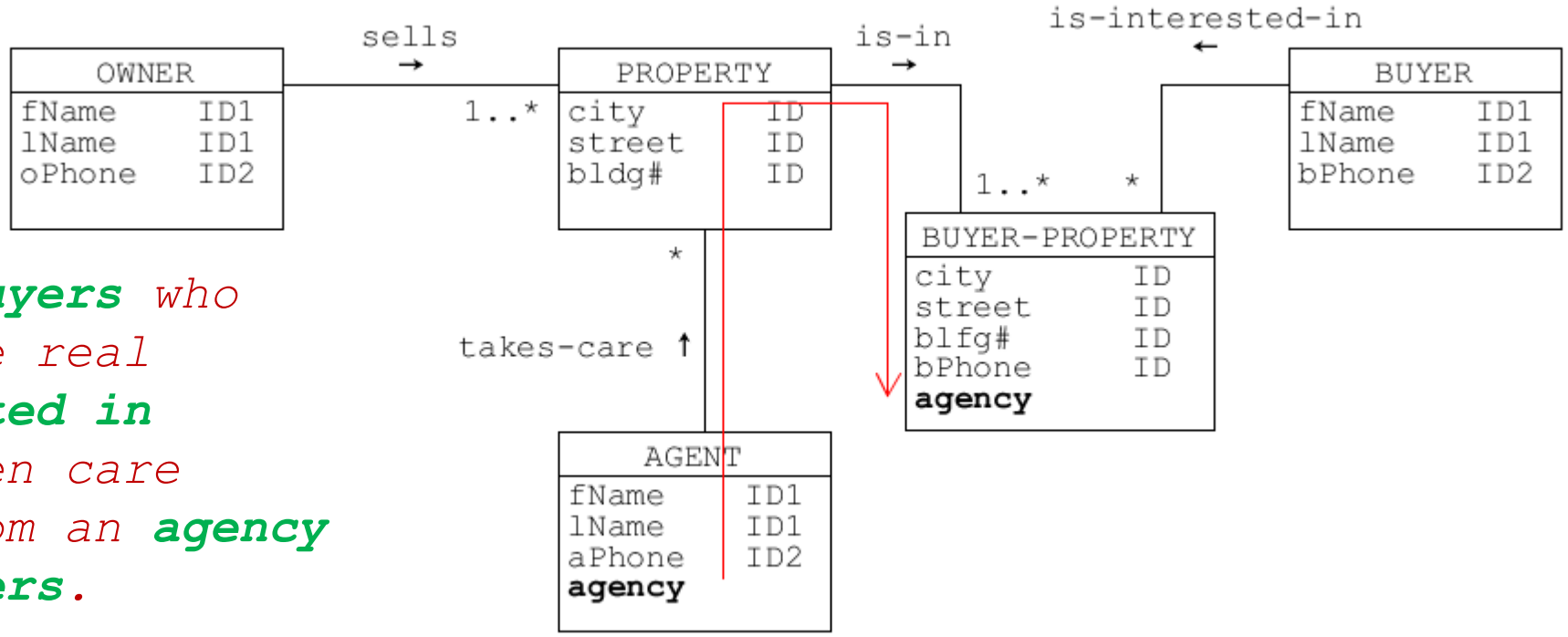
1) Perform simplification of the conceptual schema given above and redraw the simplified schema.
2) To improve performance of a class of database applications given above denormalize a conceptual schema obtained in step (1) and redraw the denormalized schema.

# 2019S1, Q1




The conceptual model is simplified by eliminating the many-to-many associations (is-interested-in).

# 2019S1, Q1




*Find the **phones of buyers** who are interested in the real estate property **located in Sydney** and being taken care about by an agent from an **agency Real Estate Demolishers**.*

The attributes bPhone and city are already in the association class ‘BUYER-PROPERTY’. To be able to answer the query with the information about the agency, we copy the attribute ‘agency’ from the class ‘AGENT’ to ‘BUYER-PROPERTY’.

# 201951, Final Examination

## Question 2 - Indexing

# 2019S1, Q2

For each one of SELECT statements listed below, find an index that speeds up the processing of a statement in the best possible way. Note that an index must be created separately for each one of SELECT statements. Write CREATE INDEX statements to create the indexes.

1) SELECT P_BRAND, COUNT(*)
   FROM PART
   GROUP BY P_BRAND
   HAVING COUNT(*) > 2;

Create index Idx1 on PART(p_brand);

An index on the attribute p_brand is created so that the system can traverse horizontally at the leaf level to compute the count.

# 2019S1, Q2

2) SELECT AVG(L_QUANTITY)
FROM ORDERS JOIN LINEITEM
ON O_ORDERKEY = L_ORDERKEY;

Create index Idx2 on LINEITEM(L_ORDERKEY, L_QUANTITY);

An index on attributes L_ORDERKEY and L_QUANTITY is created so that the system can perform a horizontal scan of the index at the leaf level and sort join the result with the relational table ORDERS to compute the aggregated average L_QUANTITY.

# 2019S1, Q2

3) SELECT AVG(PS_AVAILQTY)
FROM PARTSUPP
WHERE PS_SUPPNAME = ‘James’;

Create index Idx3 on PARTSUPP(PS_SUPPNAME, PS_AVAILQTY);

An index on the attributes PS_SUPPNAME and PS_AVAILQTY is created so that the system can perform an index range scan on the index to compute the aggregated average PS_AVAILQTY.

# 2019S1, Q2

4) SELECT P_NAME,
FROM PART
WHERE P_BRAND = ‘RUBBISH’
ORDER BY P_NAME;

CREATE index Idx4 on PART(P_BRAND, P_NAME);

An index on the attribute P_BRAND and P_NAME is created so that the system can perform an index range scan on the index to find all the P_NAME that satisfy the condition P_BRAND = ‘RUBBISH’.

# 2019S1, Q3

5) SELECT C_NAME,
FROM CUSTOMER
WHERE N_NATIONKEY = ‘SG’
MINUS
SELECT C_NAME
FROM CUSTOMER
WHERE C_ADDRESS LIKE ‘%Bugis%’;

CREATE index Idx5 on CUSTOMER(c_nationkey, c_address, c_name);

An index on the attributes C_NATIONKEY, C_ADDRESS, and C_NAME is created so that the system can perform, first time, a range scan for the first query and second time, a full index scan for the second query. The first result is then used to minus the second result to produce the required output.

# 201951, Final Examination

## Question 3 - Indexing

# 2019S1, Q3

Assume that a relational table

`PRODUCT(name, manufacturer, price, description, quality, mdate)`

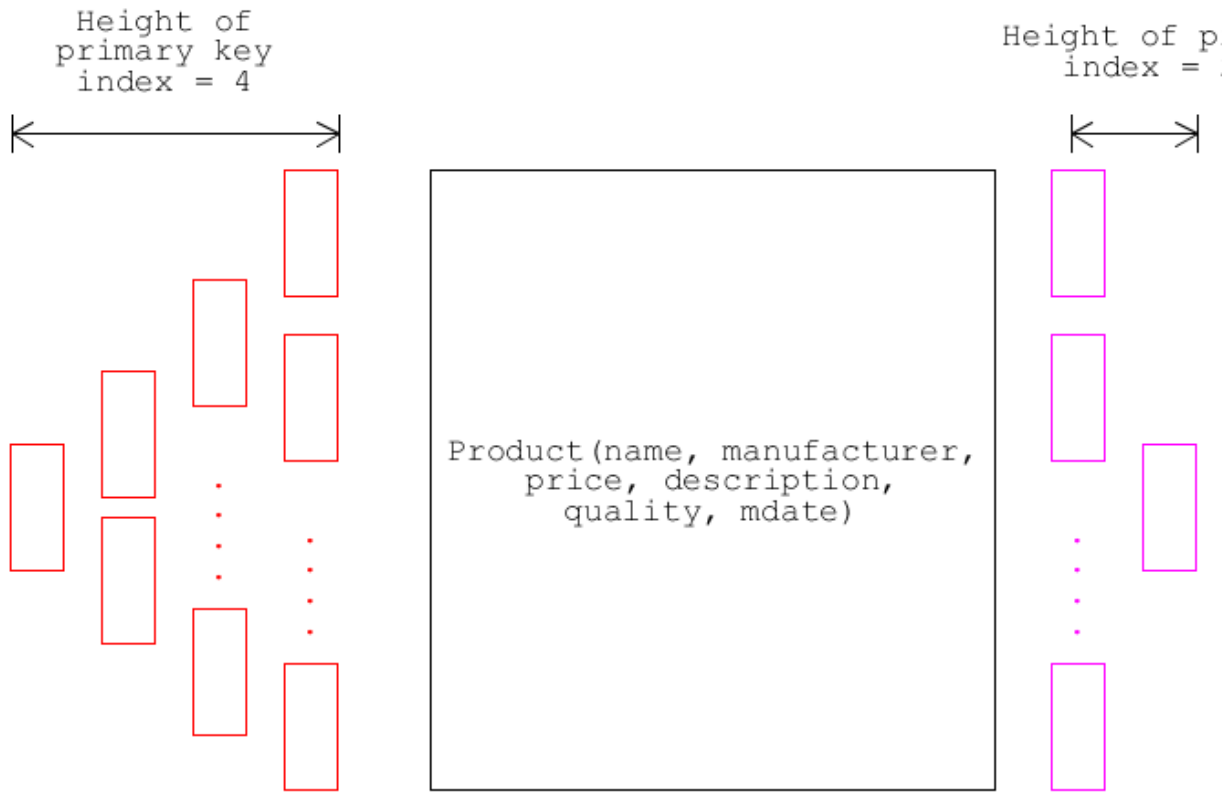
Contains information about the names, manufacturers, prices, qualities, manufacturing dates and descriptions of products. Assume that the table has a composite primary key `(name, manufacturer)`.

A database administrator created B*-Tree index on an attribute price. B*-Tree index on a primary key has been automatically created by a database system.

# 2019S1, Q3

Assume that:

i. A relational table PRODUCT occupies $10^4$ data blocks,
ii. A relational table PRODUCT contains $10^5$ rows,
iii. A height of an index on the primary key is equal to 4,
iv. A height of an index on an attribute rice is equal to 2,
v. The total number of distinct values in a column price is equal to $10^3$,
vi. A leaf level of an index on the primary key consists of 500 data blocks,
vii. A leaf level of an index on attribute price consists of 100 data blocks.




index-leaf block: 500

primary key: (name, manufacturer)

data block: 10,000

rows: 100,000

block factor = 100,000/10,000
= 10

index-leaf block: 100

distinct price = 1000

number of records per price value = 10,000/1,000
= 100

The scenario described. The scenario is drawn to aid understanding. It is not a mandatory requirement.

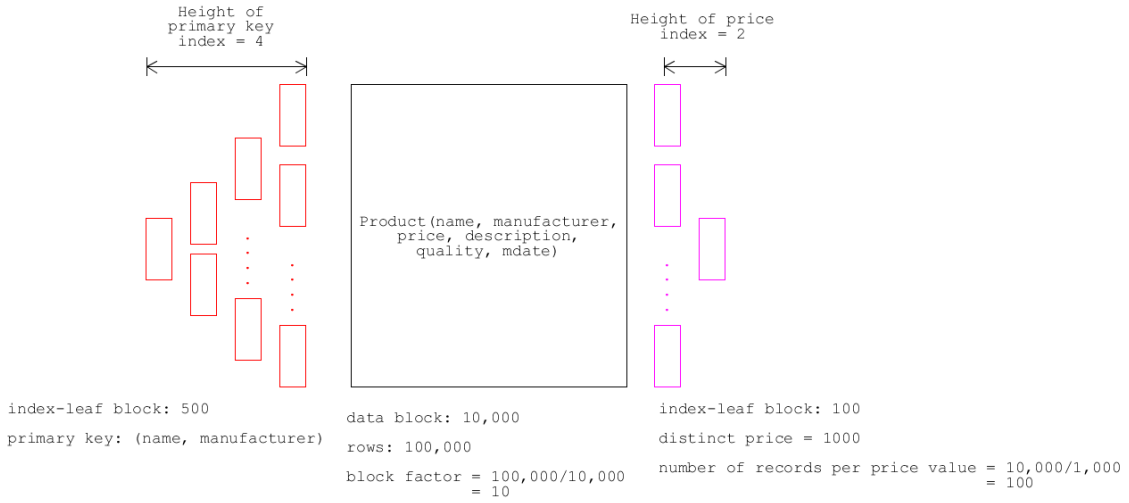
# 2019S1, Q3

List the comprehensive descriptions of query processing plans for each one of the queries listed below and estimate the total number of read block operations needed to compute each one of the queries (show all calculations):

# 2019S1, Q3

1) SELECT price, COUNT(*)
   FROM PRODUCT
   GROUP BY price;

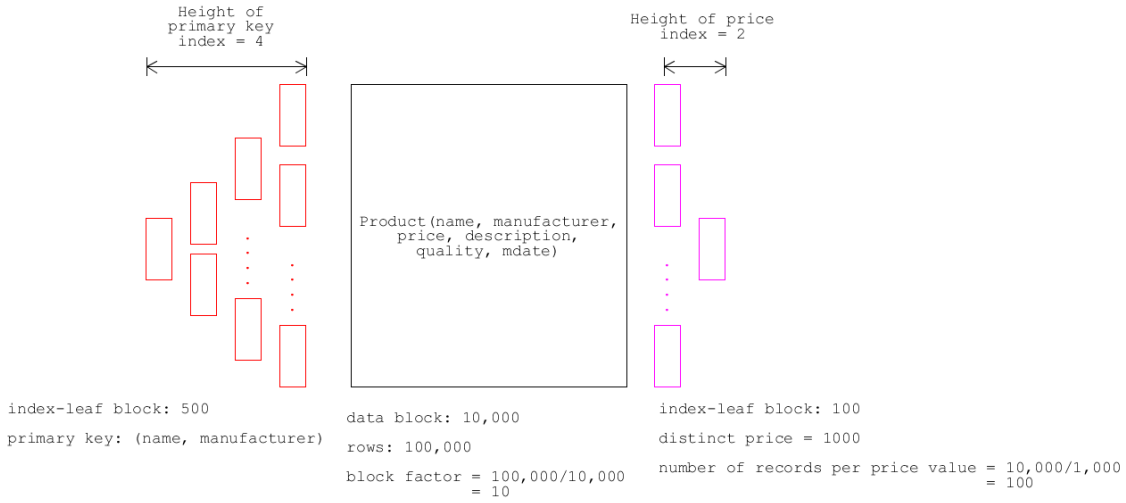
There is no ‘where’ clause in the query which implies all rows are needed. The query is computing the aggregate number of each price in the table PRODUCT, the system will perform a horizontal scan at the leaf level of the price index and compute the aggregated count for each price. Hence total number of read blocks performed is **100**.



# 2019S1, Q3

2) SELECT name, manufacturer
FROM PRODUCT
WHERE manufacturer = 'IBM' OR name = 'PC';

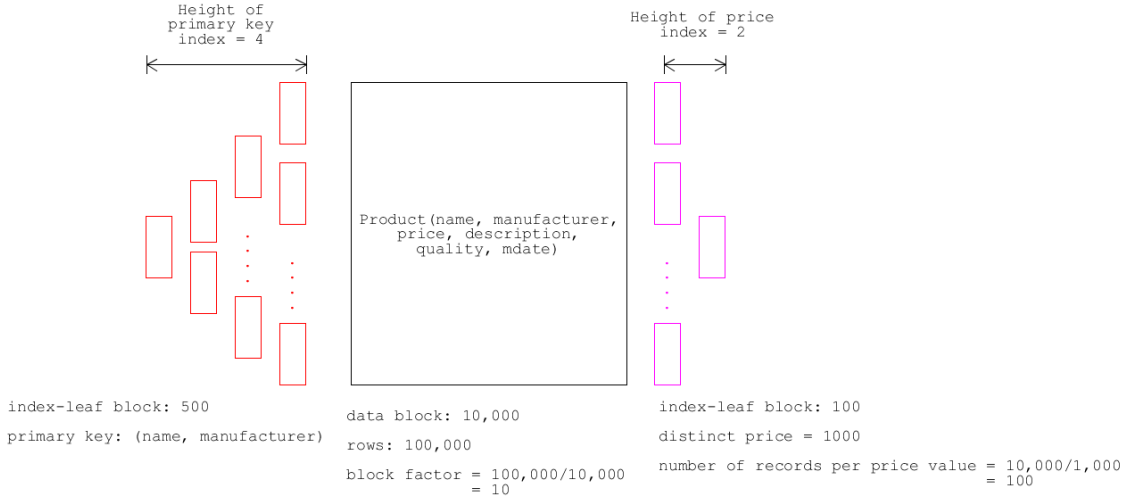
The attributes name and manufacturer can be found in the primary key index. The system will perform a horizontal scan at the leaf level of the primary key index and output the required information when the condition matched. Hence total number of read block equals **500**.



# 2019S1, Q3

3) SELECT DISTINCT price
FROM PRODUCT
ORDER BY price;

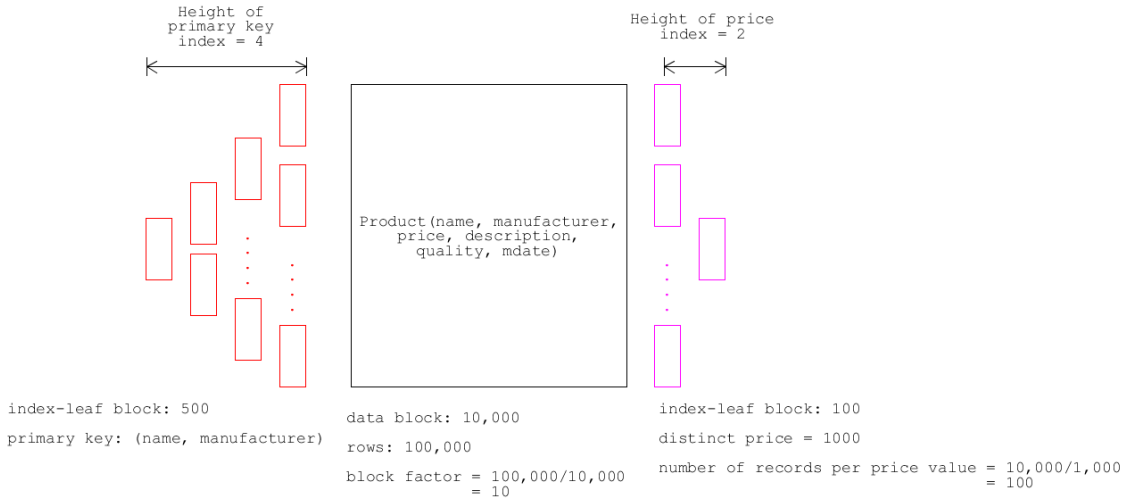
The attribute price can be found in the price index (secondary index), the system will perform a horizontal scan at the leaf level of the price index starting from the smallest price value and goes toward the largest price value, outputting the distinct value of price. Hence total number of read block equals **100**.



# 2019S1, Q3

4) SELECT *
FROM PRODUCT
WHERE manufacturer = ‘IBM’ OR name = ‘PC’;

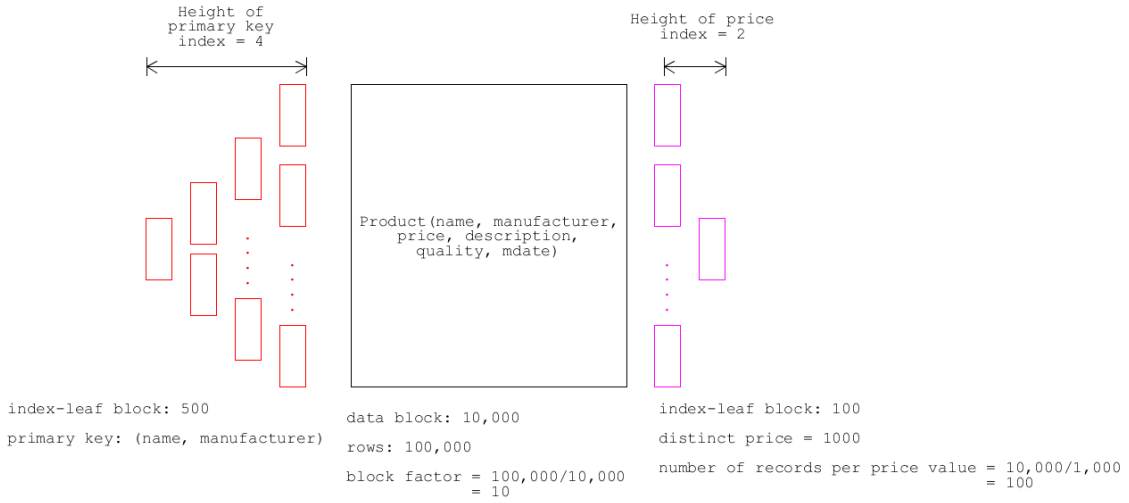
The query contains a ‘where’ clause, but it involves an ‘or’ operator and the condition manufacturer = ‘IBM’ cannot be satisfied using the primary key index, the system will perform a full-table scan to retrieve and output the required information. Hence, total number of read block equals **10,000**.



# 2019S1, Q3

5) SELECT *
FROM PRODUCT
WHERE price = 100;

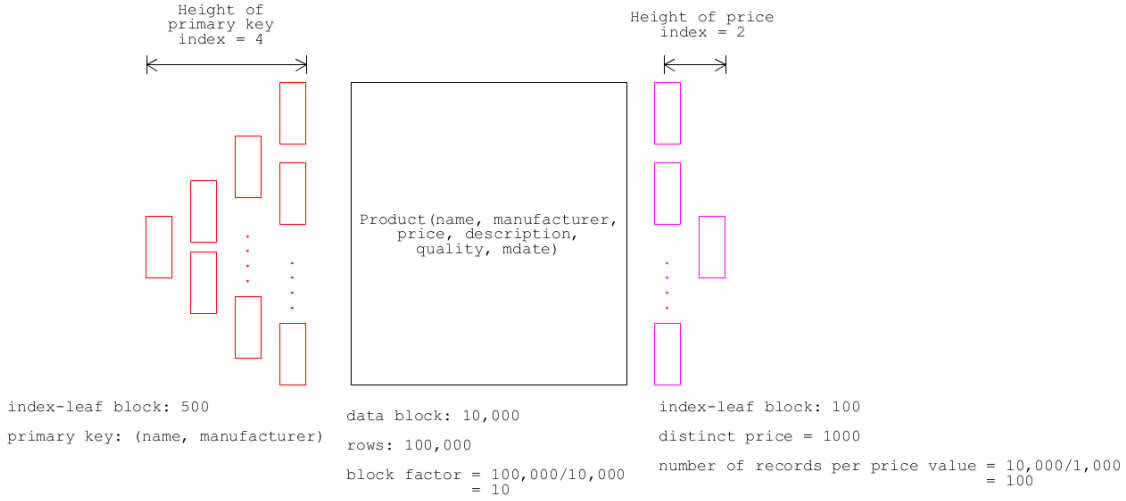
The query consists of a ‘where’ clause, and the condition price = 100 can be satisfied using the secondary index price. The system will perform a vertical scan of the index using price = 100 to reach the leaf level and following the rowid to retrieve the required rows.



# 2019S1, Q3

5) SELECT *
FROM PRODUCT
WHERE price = 100;

Since, the number of rows return is more than one (price is not unique), the system will estimate the read block based on an average case which equals (worst case + best case) / 2. Since there are 1000 distinct price values and there are 10,000 rows in product table, on the average there are about 100 records per price value.

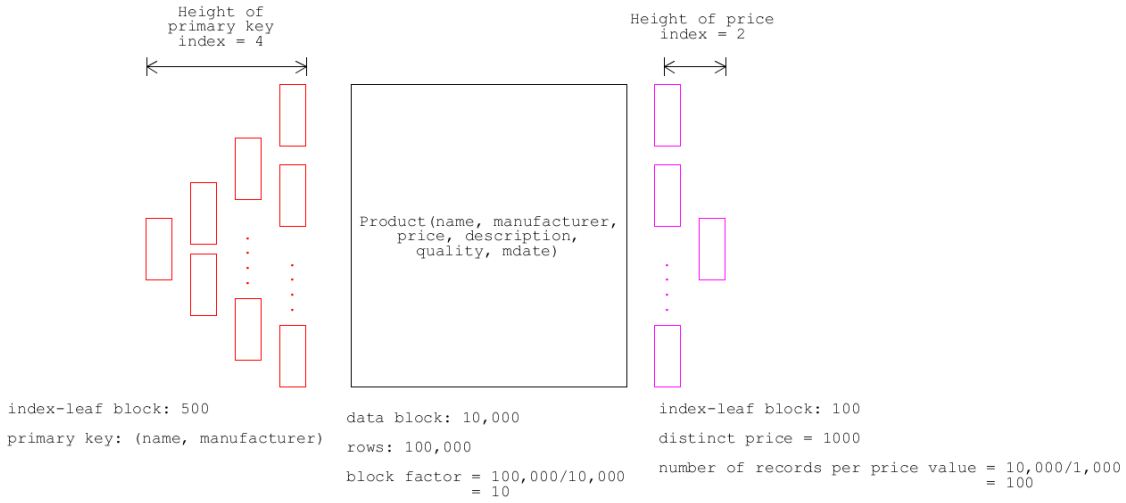


# 2019S1, Q3

5) SELECT *
FROM PRODUCT
WHERE price = 100;

The block factor of the described scenario is 100,000 rows / 10,000 blocks = 10 rows/block. The average read block is then determined as follow:

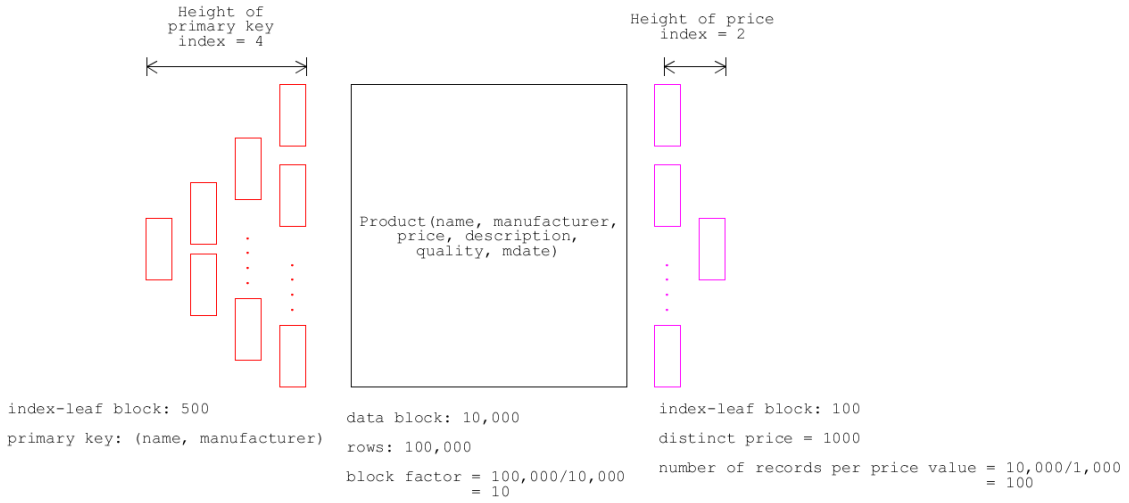
- Best case: 100 rows/10 rows per block = 10 read blocks.
- Worst case: 100 rows/1 rows per block = 100 read blocks.



# 2019S1, Q3

5) SELECT *
FROM PRODUCT
WHERE price = 100;

- Average case = (10 + 100)/2 = 55 read blocks
- Total read blocks = vertical traversal of the price index + average case = 2 + 55 = **57** read blocks.

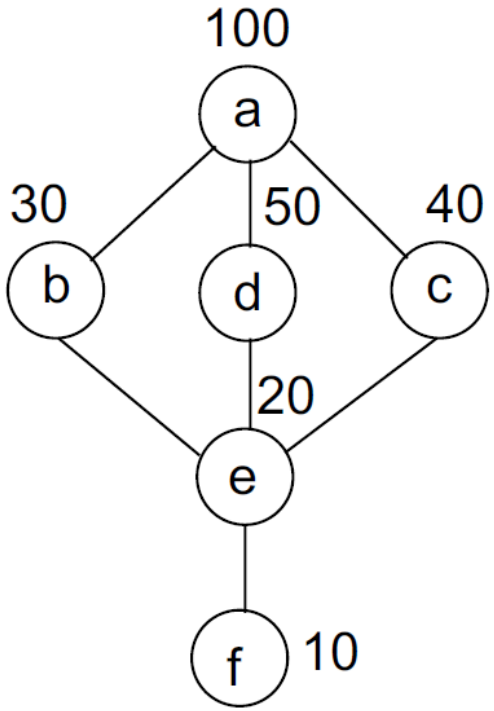


# 201951, Final Examination

## Question 4 – Materialized View

# 2019S1, Q4

Consider the following incomplete lattice of materialized views:

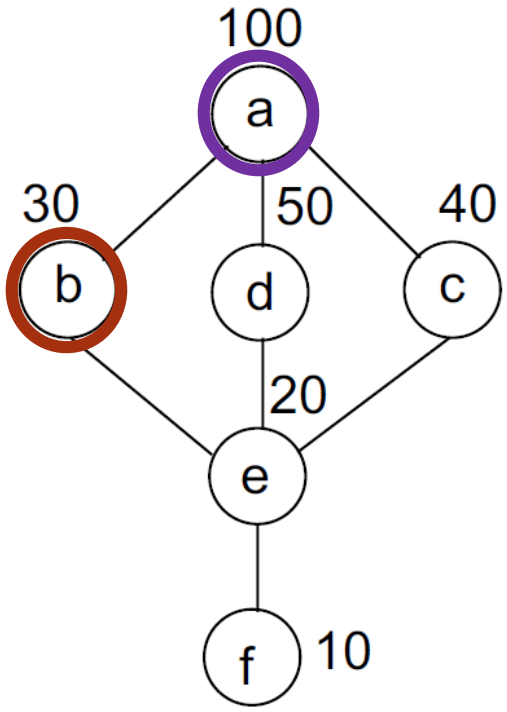



Assume that a view “a” has been already materialized and a cost of its materialization is 100. The costs of materialization of the other views “b”, “c”, “d”, “e”, and “f” are given in the diagram shown.

Use an algorithm included in a presentation 19 Materialized views to find no more than two other views that can be materialized in order to reduce the costs of view processing in the best way.

# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | 210 | | | | | |

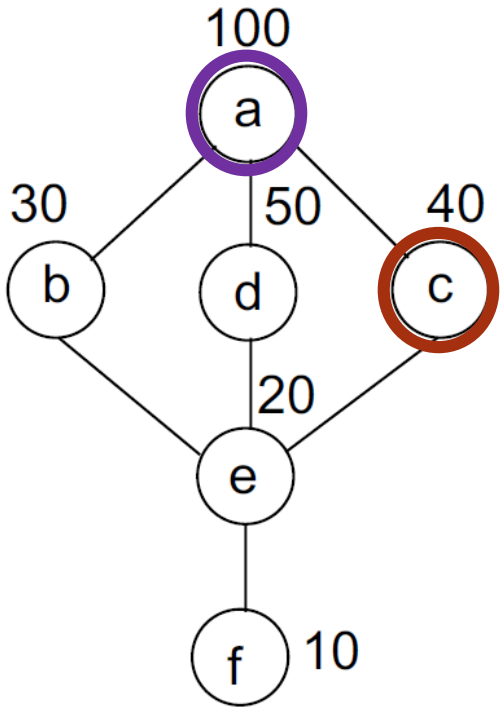



First iteration:

To materialize view b given S, we compute the benefit B(b, S) = $\sum_{w \leq a} B_b = (100 - 30) \times 3 = 210$

# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | 210 | 180 | | | | |

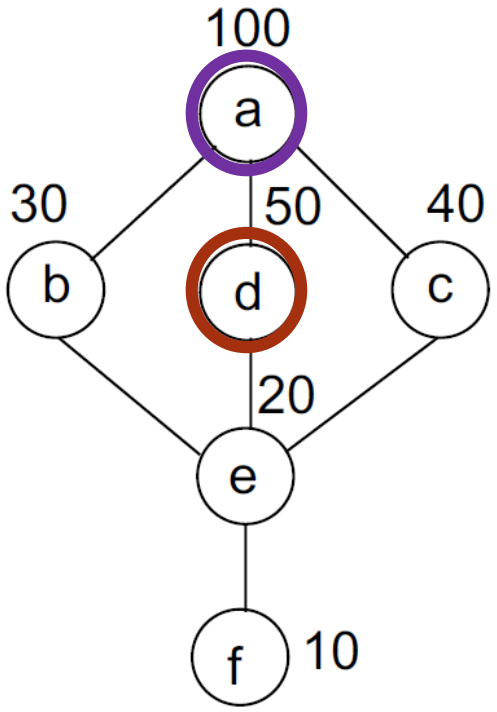



First iteration:

To materialize view c given S, we compute the benefit B(c, S) = $\sum_{w \leq c} B_c = (100 - 40) \times 3 = 180$

# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | 210 | 180 | 150 | | | |




First iteration:

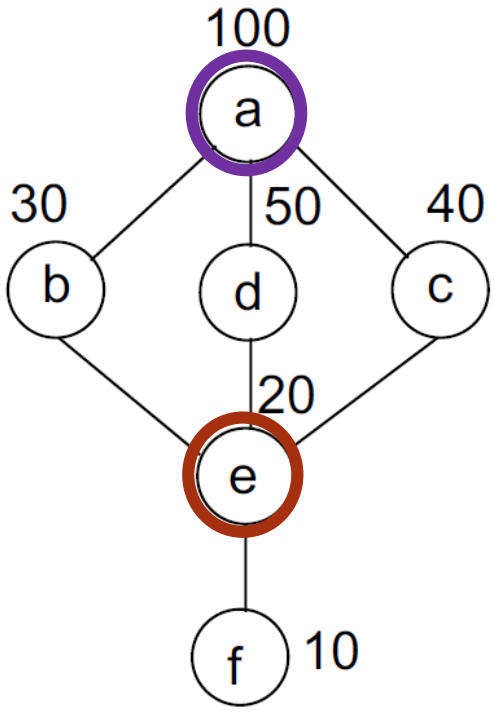
To materialize view d given S, we compute the benefit B(d, S) = $\sum_{w \leq d} B_d = (100 - 50) \times 3 = 150$

# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | 210 | 180 | 150 | 160 | | |

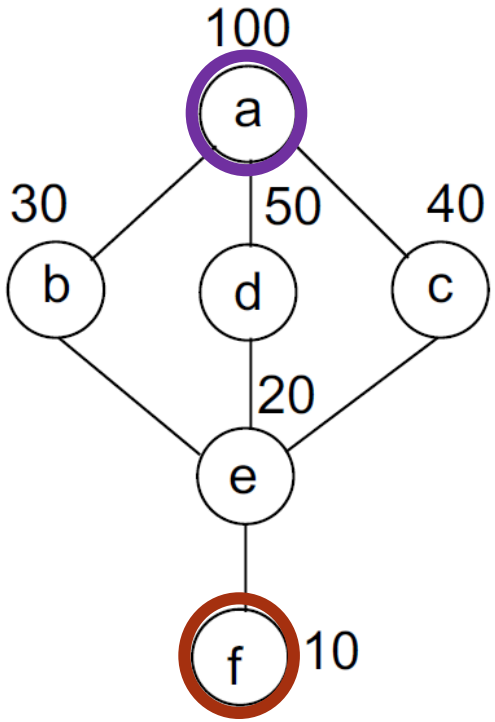
First iteration:

To materialize view e given S, we compute the benefit B(e, S) = $\sum_{w \leq e} B_e = (100 - 20) \times 2 = 160$



# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | 210 | 180 | 150 | 160 | 90 | |




First iteration:

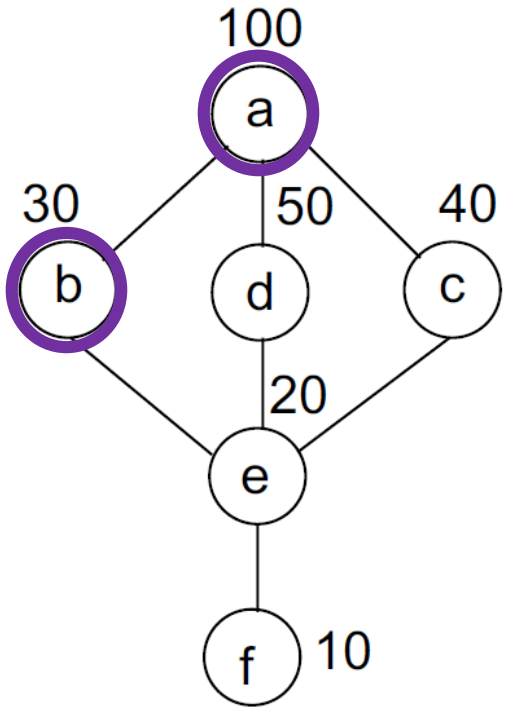
To materialize view f given S, we compute the benefit B(f, S) = $\sum_{w \leq f} B_f = (100 - 10) \times 1 = 90$

# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | 180 | 150 | 160 | 90 | |

First iteration:

From the cost reductions (benefits) computed in the first iteration, it is noted to materialize view b yield the most benefit, that is, 210. Hence view **b** is to be selected.

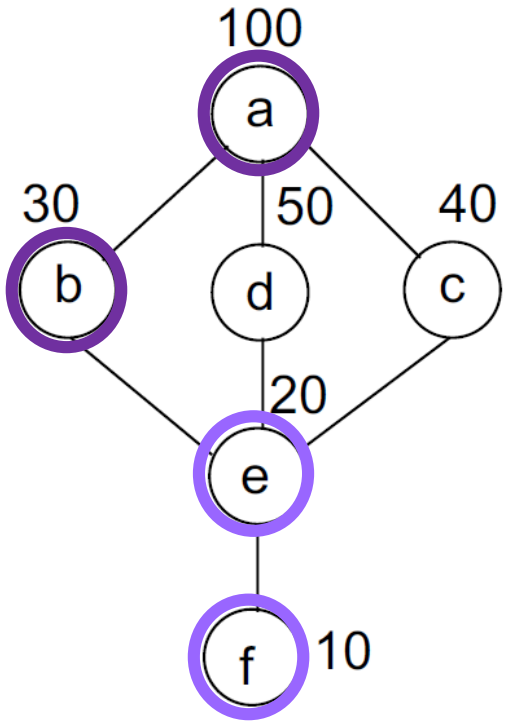


# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | | | | | |

Second iteration:

With the views **a** and **b** selected in set S, we start the second iteration to compute the benefits to realize views c, d, e and f.

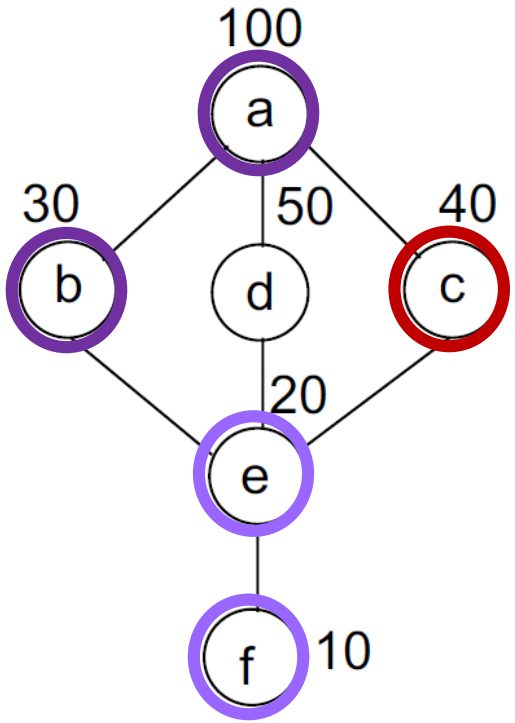


# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | 60 | | | | |

Second iteration:

To materialize view c given S, we compute the benefit B(c, S) = $\sum_{w \leq c} B_c = (100 - 40) \times 1 = 60$. Note that materialising view c only benefits itself because views e and f are benefitted through view b.

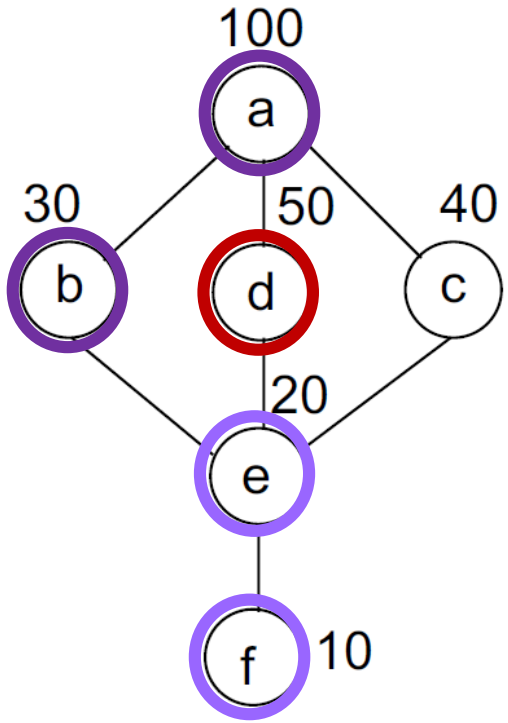


# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | 60 | 50 | | | |

Second iteration:

To materialize view d given S, we compute the benefit B(d, S) = $\sum_{w \leq d} B_d = (100 - 50) \times 1 = 50$. Similarly, materialising view d only benefits itself because views e and f are benefitted through view b.

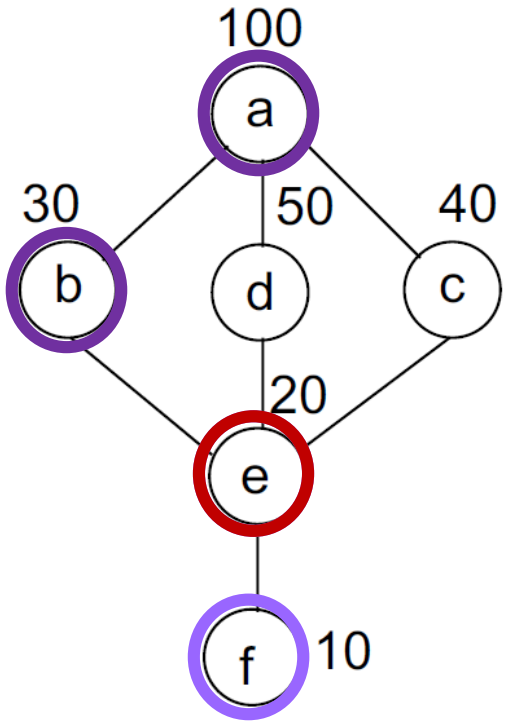


# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | 60 | 50 | 20 | | |

Second iteration:

To materialize view e given S, we compute the benefit B(e, S) = $\sum_{w \leq e} B_e = (30 - 20) \times 2 = 20$. Note that with view b, materialized, views e and f will be benefitted from view b instead of view a because view e and f are immediate decedents of view b.

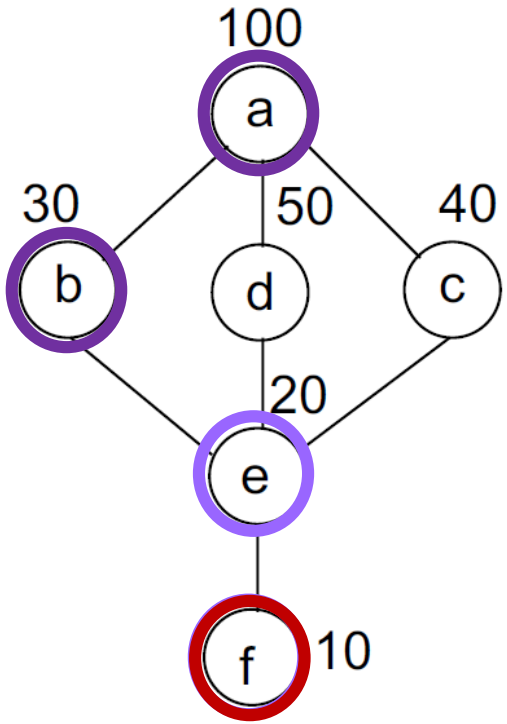


# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | 60 | 50 | 20 | 20 | |

Second iteration:

To materialize view f given S, we compute the benefit B(f, S) = $\sum_{w \leq f} B_f = (30 - 10) \times 1 = 20$. Note that with view b, materialized, views e and f will be benefitted from view b instead of view a because view e and f are immediate decedents of view b.

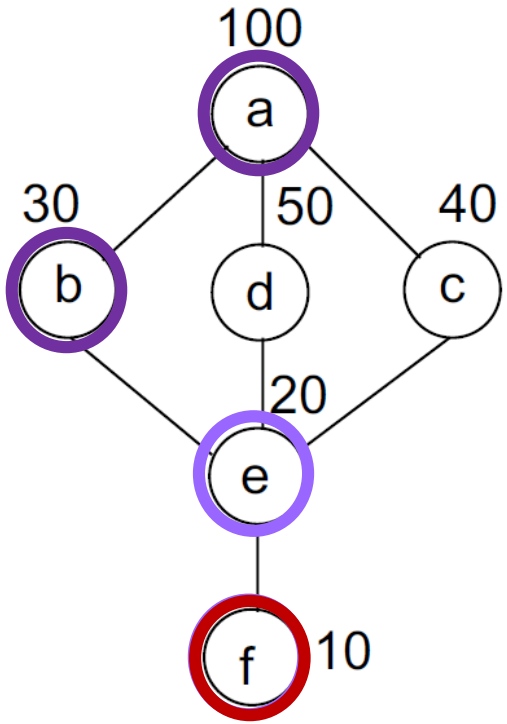


# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | **c** | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | **60** | 50 | 20 | 20 | |

Second iteration:

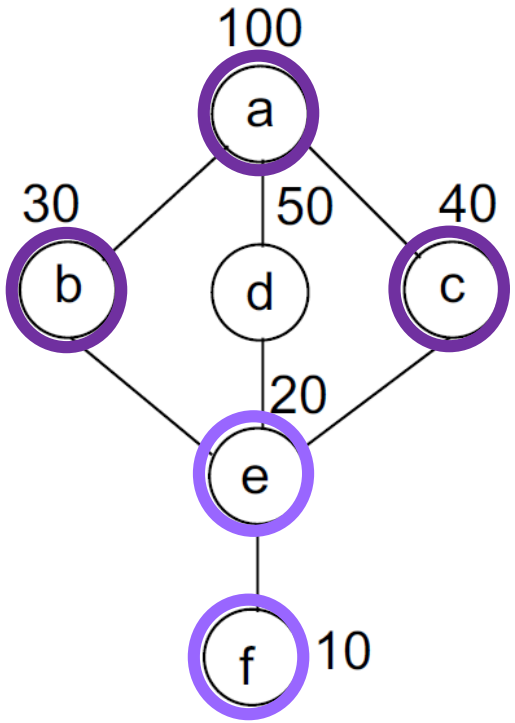
From the cost reductions (benefits) computed in the second iteration, it is noted to materialize view c yield the most benefit, that is, 60. Hence view **c** is to be selected.



# 2019S1, Q4

| | | | | | | |
|---|---|---|---|---|---|---|
| S={ | **a** | **b** | **c** | | | } |
| View to Materialized: W{ | b | c | d | e | f | } |
| Cost reduction: | **210** | **60** | 50 | 20 | 20 | |

Since the question specifies that not more than two other views are to be materialized, we stop after determining that the two views **b** and **c** are materialized.



# 201951, Final Examination

Question 5 – JDBC

# 2019S1, Q5

Consider a fragment of simple JDBC application listed below. It is a typical example of a pretty poor, from performance point of view, JDBC program. Rewrite a code written below to improve the performance of the application it is included in. There is no need to write the entire JDBC application.

Explain all details why the original application takes long time to provide the results and why your version of JDBC code is more efficient than the original one.

# 2019S1, Q5

```
try{
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery( "SELECT * FROM LINEITEM" );
      int counter = 0;
      float total; = 0.0;
      while ( rset.next() )
        {
            total = total + rset.getFloat(4);
            counter++;
      }
      System.out.println( "Result: " + total/counter );
    }
```

# 2019S1, Q5

```
LINEITEM(
L_ORDERKEY NUMBER(12) NOT NULL,
L_PARTKEY NUMBER(12) NOT NULL,
L_LINENUMBER NUMBER(12) NOT NULL,
L_QUANTITY NUMBER(12,2) NOT NULL,
L_SHIPDATE DATE NOT NULL,
CONSTRAINT LINEITEM_PKEY PRIMARY KEY
(L_ORDERKEY, L_LINENUMBER),
CONSTRAINT LINEITEM_FKEY1 FOREIGN KEY
(L_ORDERKEY)
REFERENCES ORDERS(O_ORDERKEY),
CONSTRAINT LINEITEM_FKEY2 FOREIGN KEY
(L_PARTKEY)
REFERENCES PART(P_PARTKEY) );
```

- Analysing the code, and it is noted the segment of application processes one expensive 'SELECT' statement and one loop process to compute the average quantity (the fourth attributes of LINEITEM table.)

```
try{
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery( "SELECT * FROM LINEITEM" );
      int counter = 0;
      float total; = 0.0;
      while ( rset.next() )
        {
            total = total + rset.getFloat(4);
            counter++;
      }
      System.out.println( "Result: " + total/counter );
   }
```

- First, the 'SELECT * FROM LINEITEM;' statement. The statement is sent to the dbms server which in turn returns the set of LINEITEM rows.
- Second, the 'while (rset.next() )' statement loops through the set of rows returned from the dbms for the LINEITEM.

```
try{
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery( "SELECT * FROM LINEITEM" );
      int counter = 0;
      float total; = 0.0;
      while ( rset.next() )
        {
            total = total + rset.getFloat(4);
            counter++;
      }
      System.out.println( "Result: " + total/counter );
   }
```

- Total number of quantity are accumulated and at the same time, the total number of rows is accumulated.
- Finally, after the loop is completed, the average computed as (Total/counter) is output.

```
try{
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery( "SELECT * FROM LINEITEM" );
      int counter = 0;
      float total; = 0.0;
      while ( rset.next() )
        {
            total = total + rset.getFloat(4);
            counter++;
      }
      System.out.println( "Result: " + total/counter );
   }
```

- The in efficiency of this segment of code comes from one expensive ‘SELECT’ operation and one loop. These two operations loops increase the processing at the client site (application).

```
try{
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery( "SELECT * FROM LINEITEM" );
      int counter = 0;
      float total; = 0.0;
      while ( rset.next() )
        {
            total = total + rset.getFloat(4);
            counter++;
      }
      System.out.println( "Result: " + total/counter );
   }
```

- To improve the performance of the application (client site), we can let the server, which is more powerful, to compute the average quantity of lineitem before returning the aggregated value to the application (client) to display.

```
try{
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery( "SELECT * FROM LINEITEM" );
      int counter = 0;
      float total; = 0.0;
      while ( rset.next() )
        {
            total = total + rset.getFloat(4);
            counter++;
      }
      System.out.println( "Result: " + total/counter );
    }
```

For example,

Statement stmt = conn.createStatement();

ResultSet rset = stmt.executeQuery( "SELECT AVG(L_QUANTITY) FROM LINEITEM" );

rset.next();

System.out.println("Result: " + rset.getFloat(1));

# 201951, Final Examination

## Question 6 – Execution Plan

# 2019S1, Q6

Consider the SELECT statements given below. Each one of the given SELECT statements joins two or more relational tables. For each SELECT statement propose the best method for the implementation of the join algorithm. Justify your choice! Note that answers without the exhaustive and correct justifications score no marks!

# 2019S1, Q6

Consider the following implementations of join operations:

i. Cartesian product join
ii. Nested loop join
iii. Nested loop join with one or both arguments kept in transient memory
iv. Index-based join
v. Sort-merge join
vi. Hash join
vii. Hash antijoin.

# 2019S1, Q6

1) SELECT *
   FROM ORDERS, LINEITEM
   WHERE ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY;

   The relational tables ORDERS and LINEITEM are associated via a foreign key L_ORDERKEY in the relational table LINEITEM that references the O_ORDERKEY in the relational table ORDERS. Since the number of rows in the relational table ORDERS (450,000 rows) is very much smaller than the number of rows in the relational table LINEITEM (1,800,093 rows), the system will perform a hash join by performing a full table scan on the relational table ORDERS and build a hash table based on the join key O_ORDERKEY. The system then scans the larger table LINEITEM and performs the same hashing algorithm on the attribute L_ORDERKEY. If the hash keys match, the system will output the matching rows.

# 2019S1, Q6

2) SELECT *
FROM PART JOIN PARTSUPP
ON PART.P_AMOUNT = PARTSUPP.PS_AVAILQTY;

Both the attributes used in the join condition are not indexed, and since the size of the relational table PART is smaller than the relational table PARTSUPP, the system will perform a HASH join instead of Cartesian join. The system will perform a full table scan on the relational table PART and build a hash table based on the join key P_AMOUNT. The system then scans the larger table PARTSUPP and performs the same hashing algorithm on the attribute PS_AVAILQTY. If the hash keys match, the system will output the matching row.

# 2019S1, Q6

3) SELECT *
FROM PART
WHERE EXISTS (SELECT *
FROM LINEITEM
WHERE PART.P_PARTKEY = LINEITEM.L_PARTKEY);

The query uses ‘EXISTS’ clause and hence the system will perform a SEMI join. The system will first perform a full table scan of the first table, in this case, the relational table PART. The key P_PARTKEY is then used to find a match in the second table, LINEITEM. If a match is found, a row of PART is output. Only one matched row is return. If there are other rows in the second table LINEITEM that match the condition, they will be ignored.

# 2019S1, Q6

4) SELECT *
FROM ORDERS
WHERE NOT EXISTS (SELECT *
FROM LINEITEM
WHERE ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY);

The query uses ‘NOT EXISTS’ clause and hence the system will perform a ANTI join. The system will first perform a full table scan of the first table, in this case, the relational table ORDERS. The key O_ORDERKEY is then used to find a match in the second table, LINEITEM. If there is **NO** match is found, a row of ORDERS is output.

# 2019S1, Q6

5) SELECT *
FROM LINEITEM
WHERE LINEITEM.L_QUANTITY > ( SELECT L_QUANTITY
FROM LINEITEM
WHERE L_ORDERKEY = 1 AND L_LINENUMBER = 1);

The query uses an in-equality condition '>' and hence the system will perform a sort-merge join. The system will access the relational table LINEITEM two times, the first time for the outer query using unique index scan, and the second time for the inner (sub-) query by full-table scan. Sort the second table in-memory based on the primary key order. Next, the sort-merge would merge the two sorted list based on the conditions, that is, L_QUANTITY in the first table > L_QUANTITY in the second table, and the conditions L_ORDERKEY = 1 and L_LINENUMBER = 1 of rows in second tables are satisfied.

# 201951, Final Examination

## Question 7 – Transaction Chopping

# 2019S1, Q7

Consider three database transactions given below.

Transaction 1

UPDATE PART

SET P_SIZE = P_SIZE + 1 WHERE P_PARTKEY = 1;

UPDATE PART

SET P_SIZE = P_SIZE + 2 WHERE P_PARTKEY = 2;

UPDATE PART

SET P_SIZE = P_SIZE + 3 WHERE P_PARTKEY = 3;

# 2019S1, Q7

Transaction 2

UPDATE PART

SET P_SIZE = P_SIZE + 3 WHERE P_PARTKEY = 5;

UPDATE PART

SET P_SIZE = P_SIZE + 3 WHERE P_PARTKEY = 3;

UPDATE PART

SET P_SIZE = P_SIZE + 2 WHERE P_PARTKEY = 2;

# 2019S1, Q7

Transaction 3

UPDATE PART

SET P_SIZE = P_SIZE + 4 WHERE P_PARTKEY = 4;

UPDATE PART

SET P_SIZE = P_SIZE + 23 WHERE P_PARTKEY IN (2,3);

COMMIT;

Use a technique of SC-graphs to “chop” the transactions into smaller transactions such that their concurrent processing is more efficient.

Draw SC-graph and rewrite the transaction with inserted COMMIT statements that “chop” the transactions into the smaller pieces.