ÖREBRO UNIVERSITY

Advanced SQL

Database Design - Assignment 5

Professor: Thomas Padron-McCarthy

Students: Özgun Mirtchev & Duy Dinh

2015-01-03

In this document you will find the SQL-queries in each table adjacent to the result of said query, along with potential comments. This layout makes it easier to read and understand each individual task.

1. Which part is heaviest, and what does it weigh?

INPUT

```
SELECT p.pname, p.weight
FROM parts p
WHERE p.weight = (SELECT MAX(weight)
FROM parts);
```

OUTPUT

*	PNAME	WEIGHT
1	disk drive	685

2. Write a query that finds out which items cost more than the average.

From the query "`SELECT AVG(price) FROM ITEM;`" the value 1138 was gathered (the average price of all items), which means that any value above 1138 in the column PRICE, should be correct.

INPUT

```
SELECT name, price
FROM item
WHERE price > (SELECT AVG(price)
FROM item)
ORDER BY price DESC;
```

OUTPUT

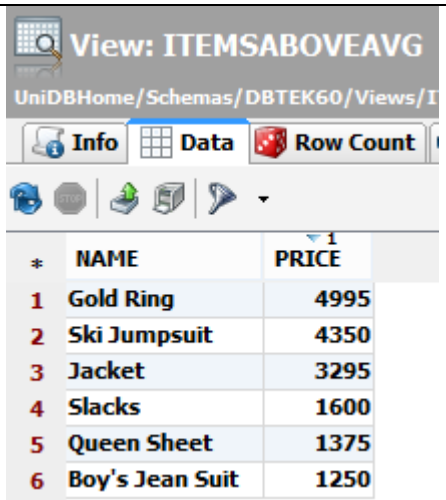
*	NAME	PRICE
1	Gold Ring	4995
2	Ski Jumpsuit	4350
3	Jacket	3295
4	Slacks	1600
5	Queen Sheet	1375
6	Boy's Jean Suit	1250

3. Create a view that contains all items that cost more than the average.

INPUT

```
CREATE VIEW ItemsAboveAvg AS
SELECT name, price
FROM item
WHERE price > (SELECT AVG(price)
FROM item)
ORDER BY price DESC;
```

OUTPUT



*	NAME	PRICE
1	Gold Ring	4995
2	Ski Jumpsuit	4350
3	Jacket	3295
4	Slacks	1600
5	Queen Sheet	1375
6	Boy's Jean Suit	1250

4. Create a new table that contains all items that cost more than the average.

INPUT

```
CREATE TABLE Items_abv_avg AS
(SELECT name, price
    FROM item
    WHERE price > (SELECT AVG(price) FROM
item));
```

OR

```
SELECT name, price
INTO Items_abv_avg
FROM item
WHERE price > (SELECT AVG(price) FROM item);
```

Working with mimer:

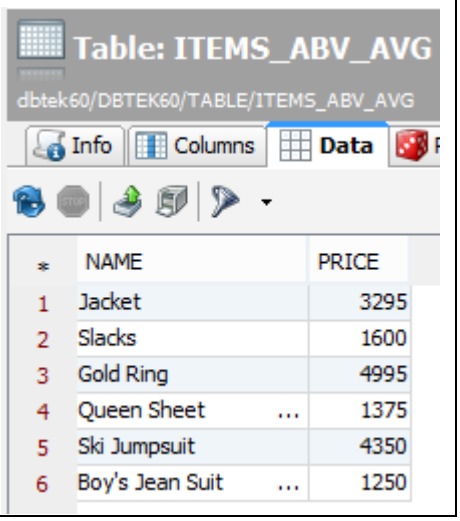
```
CREATE TABLE Items_abv_avg (
Name CHAR(30),
Price INTEGER);

INSERT INTO items_abv_avg
SELECT i.name, i.price
FROM item i
WHERE i.price > (SELECT AVG(price) FROM item);
```

OUTPUT

MimerSQL apparently doesn't allow table-creation from existing tables so the first two queries are not valid with Mimer.

The first thing that was done in this case, was creating a new table with new columns and inserting the data from the old table into the new table.



*	NAME	PRICE
1	Jacket	3295
2	Slacks	1600
3	Gold Ring	4995
4	Queen Sheet	1375
5	Ski Jumpsuit	4350
6	Boy's Jean Suit	1250

5. Explain the difference between the previous three SQL statements.

In the **first** statement, it was a plain query which asked which items cost more than the average for the user to see.

In the **second** statement, a view was created to be temporarily stored in the database for easier overlook as it automatically updates whenever the user look at it. The updates are synced from the original columns.

In the **third** statement, an attempt to create a table from the old table ITEM with the specific columns without success. To bypass that problem, a new table was created and as mentioned above, the data was inserted into the new table with the data from the old table. The table won't however update itself unless a trigger is implemented in the database to do so whenever an update occurs on a different table. This is because it has no foreign keys connecting to the other tables.

6. **Which parts have we received shipments (in the table supply) of?** ***We need the part number (pnum) and the name (pname). Write the query with a subquery in the where clause.***

INPUT	OUTPUT
`SELECT pname, pnum` `FROM parts` `WHERE pnum IN (SELECT pnum FROM supply)` `ORDER BY pnum;`	Log | 1: parts [10] × * PNAME PNUM 1 central processor 1 2 memory 2 3 disk drive 3 4 tape drive 4 5 tapes 5 6 line printer 6 7 l-p paper 7 8 terminals 8 9 terminal paper 9 10 byte-soap 10

7. **Write the same query, but this time without a subquery. Don't use an explicit join.**

INPUT	OUTPUT
`SELECT DISTINCT pname, p.pnum` `FROM parts p, supply s` `WHERE p.pnum = s.pnum` `ORDER BY p.pnum;`	Log | 1: parts [10] × * PNAME PNUM 1 central processor 1 2 memory 2 3 disk drive 3 4 tape drive 4 5 tapes 5 6 line printer 6 7 l-p paper 7 8 terminals 8 9 terminal paper 9 10 byte-soap 10

8. Write the same query, but this time with an explicit join.

INPUT

```
SELECT DISTINCT p.pname, p.pnum
FROM parts p
JOIN supply s ON s.pnum = p.pnum
ORDER BY p.pnum;
```

OUTPUT

Log | 1: parts [10]

*	PNAME	PNUM
1	central processor	1
2	memory	2
3	disk drive	3
4	tape drive	4
5	tapes	5
6	line printer	6
7	l-p paper	7
8	terminals	8
9	terminal paper	9
10	byte-soap	10

9. Which parts have we not received any shipments of? Use a subquery in the where clause.

INPUT

```
SELECT pname, pnum
FROM parts
WHERE pnum NOT IN (SELECT pnum FROM
supply)
ORDER BY pnum;
```

OUTPUT

Log | 1: parts [4]

*	PNAME	PNUM
1	card reader	11
2	card punch	12
3	paper tape reader	13
4	paper tape punch	14

10. Write the same query, but this time with an outer join.

INPUT

```
SELECT pname, p.pnum
FROM parts p
LEFT JOIN supply s
ON p.pnum = s.pnum
WHERE s.pnum IS NULL;
```

OUTPUT

Log | 1: parts [4]

*	PNAME	PNUM
1	card reader	11
2	card punch	12
3	paper tape reader	13
4	paper tape punch	14

11. How many items have been sold by each department? It is enough to just show the department number and the number of items.

INPUT

```
SELECT dept,
SUM(quantity) AS sold_items
FROM sale
GROUP BY dept;
```

OUTPUT

Log | 1: sale [6]

*	DEPT	SOLD_ITEMS
1	10	2
2	14	1
3	26	6
4	49	2
5	58	1
6	60	1

12. The same query, but now we also want the department name in the result. Write the query without an explicit join.

INPUT

```
SELECT s.dept, d.name AS dept_name,
SUM(s.quantity) AS sold_items
FROM sale s, dept d
WHERE s.dept = d.number
GROUP BY s.dept, d.name;
```

OUTPUT

Log | 1: sale [6]

*	DEPT	DEPT_NAME	SOLD_ITEMS
1	10	Candy	2
2	14	Jewelry	1
3	26	Linens	6
4	49	Toys	2
5	58	Men's	1
6	60	Sportswear	1

13. Write the same query, but this time with an explicit join.

INPUT

```
SELECT s.dept, d.name,
SUM(s.quantity) AS sold_items
FROM sale s
LEFT JOIN dept d
ON s.dept = d.number
GROUP BY s.dept, d.name;
```

OUTPUT

Log | 1: sale [6]

*	DEPT	NAME	SOLD_ITEMS
1	10	Candy	2
2	14	Jewelry	1
3	26	Linens	6
4	49	Toys	2
5	58	Men's	1
6	60	Sportswear	1

14. Write the same query once again, with the difference that departments that haven't sold any items should be in the result, with null as the number of items they have sold.

INPUT

```
SELECT d.number,
d.name AS dept_name,
SUM(s.quantity) AS sold_items
FROM dept d
LEFT JOIN sale s
ON s.dept = d.number
GROUP BY d.number, d.name;
```

OUTPUT

Log | 1: dept [19]

*	NUMBER	DEPT_NAME	SOLD_ITEMS
1	1	Bargain	(null)
2	10	Candy	2
3	14	Jewelry	1
4	19	Furniture	(null)
5	20	Major Appliances	(null)
6	26	Linens	6
7	28	Women's	(null)
8	34	Stationary	(null)
9	35	Book	(null)
10	43	Children's	(null)
11	47	Junior Miss	(null)
12	49	Toys	2
13	58	Men's	1
14	60	Sportswear	1
15	63	Women's	(null)
16	65	Junior's	(null)
17	70	Women's	(null)
18	73	Children's	(null)
19	99	Giftwrap	(null)

15. The same query as above, but now departments that haven't sold any items should have zero as the number of items they have sold. (Hint: coalesce)

INPUT

```
SELECT d.number,
d.name AS dept_name,
COALESCE(SUM(s.quantity), 0) AS
sold_items
FROM dept d
LEFT JOIN sale s
ON s.dept = d.number
GROUP BY d.number, d.name;
```

Coalesce(x1, x2):

```
CASE
WHEN x1 IS NOT NULL THEN x1 ELSE x2
END
```

OUTPUT

Log | 1: dept [19]

*	NUMBER	DEPT_NAME	SOLD_ITEMS
1	1	Bargain	0
2	10	Candy	2
3	14	Jewelry	1
4	19	Furniture	0
5	20	Major Appliances	0
6	26	Linens	6
7	28	Women's	0
8	34	Stationary	0
9	35	Book	0
10	43	Children's	0
11	47	Junior Miss	0
12	49	Toys	2
13	58	Men's	1
14	60	Sportswear	1
15	63	Women's	0
16	65	Junior's	0
17	70	Women's	0
18	73	Children's	0
19	99	Giftwrap	0

16. What is the name and the number of the department that has sold the greatest number of items? (Hint: Define a view, and use it in the query.)

INPUT

Creating the view(previous query):

```
CREATE VIEW dept_sales AS
SELECT d.number,
d.name AS dept_name,
COALESCE(SUM(s.quantity), 0) AS
sold_items
FROM dept d
LEFT JOIN sale s
ON s.dept = d.number
GROUP BY d.number, d.name;
```

Query:

```
SELECT ds.number, ds.dept_name
FROM dept_sales ds
WHERE ds.sold_items = (SELECT
MAX(sold_items) FROM dept_sales ds);
```

OUTPUT

Log | 1: dept_sales [1]

*	NUMBER	DEPT_NAME
1	26	Linens

17. Earthquake! California sinks into the ocean, and all our suppliers in California disappear under the water. Write a query to delete them from the database. What happens when you run the query, if you have declared a foreign key? What happens if you have not declared a foreign key?

INPUT	OUTPUT
`` DELETE FROM supplier s WHERE s.state = 'Calif'; ```	Upon trying deletion with a declared foreign key(FK) the client raises an error: ``` [DELETE - 0 row(s), 0.024 secs] [Error Code: -10106, SQL State: 23000] Referential constraint DBTEK60.FK_ITEM_SUPPLIER violated UPDATE/DELETE operation not valid for table DBTEK60.SUPPLIER ``` There needs to be a rule/action/trigger which activates when a DELETE query is run so the rows that are connected to the "main"-table with the FK will also be deleted with the cascading effect, or that the rule sets the connected values to a default/null value. If there was **no** FK, the only rows that would become deleted would be on the table that was chosen (supplier). The other tables who refers to the value, in this case 'Calif', would still remain intact and present inaccurate data since 'Calif' no longer exists. The database would be inconsistent.

18. Assume that you *didn't* have any foreign key declarations, and deleted the suppliers in California. Comment on the results of these two queries:

```
SELECT item.number,
item.name
FROM item
ORDER BY item.number;
```

There would be no problem to run this query as it only depends on the data from one table, in this case 'item'.

```
SELECT item.number,
item.name,
supplier.name
FROM item, supplier
WHERE item.supplier =
supplier.number
ORDER BY item.number;
```

If we run the query(with 'Calif' still remaining) we get this:
(For the purpose of the exercise, the columns have been renamed and two new columns have been added to make it easier to see where 'Calif' is).

Log | 1: item [20]

*	ITEM_NR	ITEM_NAME	S_NAME	S_NUM	S_STATE
1	19	Bellbottoms	Levi-Strauss	33	Calif
2	165	Jean	Levi-Strauss	33	Calif
3	258	Shirt	Levi-Strauss	33	Calif
4	301	Boy's Jean Suit	Levi-Strauss	33	Calif
5	26	Earrings	Koret	199	Calif
6	115	Gold Ring	Koret	199	Calif
7	23	1 lb Box	Whitman's	42	Colo
8	25	2 lb Box, Mix	Whitman's	42	Colo
9	11	Wash Cloth	Cannon	213	Ga
10	118	Towels, Bath	Cannon	213	Ga
11	120	Twin Sheet	Cannon	213	Ga
12	121	Queen Sheet	Cannon	213	Ga
13	43	Maze	Fisher-Price	89	Mass
14	107	The 'Feel' Book	Fisher-Price	89	Mass
15	119	Squeeze Ball	Fisher-Price	89	Mass
16	52	Jacket	White Stag	15	Neb
17	101	Slacks	White Stag	15	Neb
18	122	Ski Jumpsuit	White Stag	15	Neb
19	21	ABC Blocks	Playskool	125	Tex
20	106	Clock Book	Playskool	125	Tex

As we can see about a third of the table consists of items from 'Calif'. Now, the scenario is that there are no foreign keys. That means, when 'Calif' is deleted from the supplier table, the rows of data only deleted on that table, but not in the item-table, which means it still had the supplier-numbers of 'Calif', 33 and 199, remaining.

In the WHERE-clause it tries to match the same values of the two tables. In this case, since 33 and 199 are deleted from the supplier table but still remains in the item table, they will not be displayed when running this query, because there's no match on the other table. This means that everything else on the above table will be displayed, except the rows with 'Calif' in them. See picture on the next page.

This is the table when 'Calif' doesn't exist in the supplier table and there are no foreign keys.

(To get this table following query was used):

```
SELECT i.number as item_nr,
i.name as item_name,
s.name as s_name,
s.number as s_num,
s.state as s_state
FROM item i, supplier s
WHERE i.supplier = s.number
ORDER BY s.state;
```

Log | 1: item [14]

*	ITEM_NR	ITEM_NAME	S_NAME	S_NUM	S_STATE
1	23	1 lb Box	Whitman's	42	Colo
2	25	2 lb Box, Mix	Whitman's	42	Colo
3	11	Wash Cloth	Cannon	213	Ga
4	118	Towels, Bath	Cannon	213	Ga
5	120	Twin Sheet	Cannon	213	Ga
6	121	Queen Sheet	Cannon	213	Ga
7	43	Maze	Fisher-Price	89	Mass
8	107	The 'Feel' Book	Fisher-Price	89	Mass
9	119	Squeeze Ball	Fisher-Price	89	Mass
10	52	Jacket	White Stag	15	Neb
11	101	Slacks	White Stag	15	Neb
12	122	Ski Jumpsuit	White Stag	15	Neb
13	21	ABC Blocks	Playskool	125	Tex
14	106	Clock Book	Playskool	125	Tex

This is the item-table with the same situation as above, but since the supplier-numbers still remain intact in the item-table it causes inconsistency in the database since the supplier-numbers 199 and 33 does not exist. A rule should be used to change the values to NULL or some other default value.

Table: ITEM

dbtek60/DBTEK60/TABLE/ITEM

Info | Columns | Data | Row Count | Primary Key | Index

*	NUMBER	NAME	DEPT	PRICE	QOH	SUPPLIER
1	11	Wash Cloth	1	75	575	213
2	118	Towels, Bath	26	250	1000	213
3	120	Twin Sheet	26	800	750	213
4	121	Queen Sheet	26	1375	600	213
5	26	Earrings	14	1000	20	199
6	115	Gold Ring	14	4995	10	199
7	21	ABC Blocks	1	198	405	125
8	106	Clock Book	49	198	150	125
9	43	Maze	49	325	200	89
10	107	The 'Feel' Book	35	225	225	89
11	119	Squeeze Ball	49	250	400	89
12	23	1 lb Box	10	215	100	42
13	25	2 lb Box, Mix	10	450	75	42
14	19	Bellbottoms	43	450	600	33
15	165	Jean	65	825	500	33
16	258	Shirt	58	650	1200	33
17	301	Boy's Jean Suit	43	1250	500	33
18	52	Jacket	60	3295	300	15
19	101	Slacks	63	1600	325	15
20	122	Ski Jumpsuit	65	4350	125	15

19. The queries 1-16 above are used frequently in the database. We expect the database to grow to a more realistic size, with many thousands of items and many millions of sales. Which indexes should be created? Show the create index commands that should be used! (Assume that the database manager doesn't automatically create indexes on declared primary keys, so you'll have to explicitly create indexes for them too.)

```
-- PARTS TABLE
CREATE INDEX parts_number ON parts(pnum) --Primary key
CREATE INDEX parts_weight ON parts(weight ASC)
```

```
-- ITEM TABLE
CREATE INDEX item_id ON item(number) -- Primary key
CREATE INDEX item_price ON item(price ASC)
CREATE INDEX item_supplier ON item(supplier ASC)
```

```
-- SUPPLY TABLE
-- The composite primary key is indexed separately since only pnum was
used in a query.
CREATE INDEX supply_parts_number ON supply(pnum) -- Primary key
CREATE INDEX supply_j_number ON supply(jnum) -- Primary key
```

```
-- SALE TABLE
CREATE UNIQUE INDEX sale_number_item_PK ON sale(number, item) -- Composite
primary key
CREATE INDEX sale_department ON sale(dept ASC)
```

```
-- DEPT TABLE
CREATE INDEX department_number ON dept(number) -- Primary key
CREATE INDEX department_name ON dept(name ASC)
```

```
-- SUPPLIER TABLE
-- Indexing this is questionable since it's a really small table and it's
rarely used
CREATE INDEX supplier_id ON supplier(number) -- Primary key
CREATE INDEX supplier_state ON supplier(state ASC)
```

```
-- DEPT_SALES VIEW
-- Probably unnecessary as well  since it was only used in a query once
CREATE INDEX sold_department_items ON dept_sales(sold_items DESC)
```

20. Start two BSQL instances beside each other, to login twice in the same database and run two concurrent transactions. Show the effect of commit and rollback, and what happens if the two transactions try to commit conflicting changes.

To explain each transaction an explanation box has been inserted below the transaction. This will hopefully make it easier to understand what each transaction and step is doing. Most of the transactions has been taken from the example. (To separate the two instances from each other, the 1st has the queries written in uppercase letters and the 2nd instance has the queries written in lowercase letters). Both instances were logged in to the same database server and the table that was chosen for this purpose was *store*.

<table>
<tr><th>#</th><th>Transaction 1</th><th>Transaction 2</th></tr>
<tr><td rowspan="2">1</td><td>

```
SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  3 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  3 rows found
```

</td></tr>
<tr><td colspan="2">In the first step we can see that the only thing that is done is selecting the tables from each running instance.</td></tr>
<tr><td rowspan="2">2</td><td>

```
SQL>INSERT INTO store VALUES (3,
'Los Angeles', 'Calif');
SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  4 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  4 rows found
```

</td></tr>
<tr><td colspan="2">Here we are inserting a row to show an example that searching from the second instance gives the same result as the first one.</td></tr>
<tr><td rowspan="2">3</td><td>

```
SQL>START TRANSACTION;
SQL>INSERT INTO store VALUES
(2,'New York', 'NY');
SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  5 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  4 rows found
```

</td></tr>
<tr><td colspan="2">Now a transaction is started to demonstrate the isolation of that transaction from the other instance. The ‘New York’ row is inserted within the transaction of the first instance. But since it hasn’t been committed yet, it will not show up on the 2nd instance using the search-query.</td></tr>
</table>

<table>
<tr><th>#</th><th>Transaction 1</th><th>Transaction 2</th></tr>
<tr><td rowspan="2">4</td><td>

```
SQL>COMMIT;
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                5 rows found
```

</td></tr>
<tr><td colspan="2">When we commit the transaction, it will now show up on the 2nd instance since the transaction has ended.</td></tr>
<tr><td rowspan="2">5</td><td>

```
SQL>START TRANSACTION;
SQL>INSERT INTO store VALUES (1,
'Chicago', 'Illi');
```

</td><td>

```
SQL>start transaction;
SQL>insert into store values (4,
'Minneapolis','Minnes');
```

</td></tr>
<tr><td colspan="2">Here we start two parallel transactions to insert values into the same table.</td></tr>
<tr><td rowspan="2">6</td><td>

```
SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          1 Chicago         Illi
          2 New York        NY
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                6 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          4 Minneapolis     Minnes
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                6 rows found
```

</td></tr>
<tr><td colspan="2">As we can see, the queries have been run for each instance, but each instance has their own values from the insert. The 1st instance has Chicago, of which the 2nd does not and the 2nd instance has Minneapolis, of which the 1st has not. This is another demonstration of the isolation of each transaction, as they cannot see each other updates, but only update the table that was in the original state before the transactions.</td></tr>
<tr><td rowspan="2">7</td><td>

```
SQL>ROLLBACK;
SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                5 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          4 Minneapolis     Minnes
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                6 rows found
```

</td></tr>
<tr><td colspan="2">The 1st instance got a rollback, which means that all the changes that were made during the transaction gets reverted and are not applied to the real database table. This means that the row that contained Chicago, is removed.
The 2nd instance is still in a transaction that has not been committed, so the updated values has not been applied the database, which is why the 1st instance still won't show the number 4 row.</td></tr>
</table>

<table>
<tr><th>#</th><th>Transaction 1</th><th>Transaction 2</th></tr>
<tr><td>8</td><td>

```
SQL>START TRANSACTION;
SQL>INSERT INTO store VALUES (4,
'Atlanta', 'Georg');
SQL>INSERT INTO store VALUES (6,
'San Diego', 'Calif');

SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          4 Atlanta         Georg
          5 San Francisco   Calif
          6 San Diego       Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  7 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          4 Minneapolis     Minnes
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  6 rows found
```

</td></tr>
<tr><td></td><td colspan="2">Now the 1st instance start a transaction, and inserts two rows instead of one. This time, an insertion on the same row will occur. That means, that the 1st instance will insert a number-4 row, whereas the 2nd instance already has a number-4 row but still hasn't committed, so it's still in a transaction state.</td></tr>
<tr><td>9</td><td>

```
SQL>COMMIT;
SQL>SELECT * FROM store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          4 Atlanta         Georg
          5 San Francisco   Calif
          6 San Diego       Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  7 rows found
```

</td><td>

```
SQL>select * from store;
     NUMBER CITY            STATE
========== =============== ======
          2 New York        NY
          3 Los Angeles     Calif
          4 Minneapolis     Minnes
          5 San Francisco   Calif
          7 Oakland         Calif
          8 El Cerrito      Calif

                  6 rows found
```

</td></tr>
<tr><td></td><td colspan="2">Now a commit is executed from the 1st instance to apply the changes to the database table. As we can see the table has been updated and contains the inserted rows from the transaction. The 2nd instance still hasn't committed its changes and still have the table that's in the transaction state.</td></tr>
</table>

<table>
<tr><th>#</th><th>Transaction 1</th><th>Transaction 2</th></tr>
<tr><td rowspan="2">10</td><td></td><td>SQL>commit;

Mimer SQL error -10001 in function EXECUTE
Transaction aborted due to conflict with other transaction</td></tr>
<tr><td colspan="2">When we commit the 2nd instance, it will come up with an error, saying that it conflicted with another transaction. This means that it tried to insert values into the same row. The 1st instance inserted ‘Atlanta’ into the number 4 row, whereas the 2nd instance transaction inserted ‘Minneapolis’ on the same row. Since the 1st instance committed before the 2nd one, its changes got applied first. The 2nd instance committed after the 1st instance and the system noticed that a recent transaction had already inserted a row into the same row as the 2nd instance transaction was trying to do. It raised an error and executed a rollback to revert all the changes that were made in the 2nd instance because of atomicity, which says that if a transaction is to be committed, the whole transaction has to be completed, or else nothing will. Which is why it reverted all the made changes and aborted the transaction.

Had we committed the 2nd instance before the 1st instance, it would’ve also given an error (to the 1st instance), but instead it would’ve applied the changes made in the 2nd instance to the database. So if the query below was run it would’ve not showed the current tables, but instead number-4 row would’ve had ‘Minneapolis’ in it and number-6 row with ‘San Diego’ would not exist.</td></tr>
<tr><td rowspan="2">11</td><td>SQL>SELECT * FROM store;
NUMBER CITY STATE
========== ========== ======
2 New York NY
3 Los Angeles Calif
4 Atlanta Georg
5 San Francisco Calif
6 San Diego Calif
7 Oakland Calif
8 El Cerrito Calif

7 rows found</td><td>SQL>select * from store;
NUMBER CITY STATE
========== ========== ======
2 New York NY
3 Los Angeles Calif
4 Atlanta Georg
5 San Francisco Calif
6 San Diego Calif
7 Oakland Calif
8 El Cerrito Calif

7 rows found</td></tr>
<tr><td colspan="2">So now when we search from both instances, we can see that only the updates from the 1st instance were applied to the database table, since each instance shows identical tables.</td></tr>
</table>

Let’s do another example of conflicting transactions on the next page, where we delete some rows in one instance and try to update the same row from another instance.

#	Transaction 1	Transaction 2
1	SQL>start transaction; SQL>update store set state = 'Cal' where number = 6; 1 row updated SQL>SELECT * FROM store; NUMBER CITY STATE ========== ========== ====== 2 New York NY 3 Los Angeles Calif 4 Atlanta Georg 5 San Francisco Calif 6 San Diego Cal 7 Oakland Calif 8 El Cerrito Calif 7 rows found	SQL>start transaction; SQL>delete from store where number = 2; 1 row deleted SQL>delete from store where number = 6; 1 row deleted SQL>commit; SQL>select * from store; NUMBER CITY STATE ========== ========== ====== 3 Los Angeles Calif 4 Atlanta Georg 5 San Francisco Calif 7 Oakland Calif 8 El Cerrito Calif 5 rows found
	So here we have the table from the previous step. In the 1st instance, an update to the state-name of a row was made but wasn't committed yet. In the 2nd instance, two delete queries were executed AND committed. Now the current table looks like the one in the 2nd instance. Let's see what happens when we commit the 1st instance transaction.	
2	SQL>commit; Mimer SQL error -10001 in function EXECUTE Transaction aborted due to conflict with other transaction	
	Once again, as one can see, the client raised an error stating that it was conflicting with another transaction. Since we committed the 2nd instance before the 1st instance, it wouldn't know that the rows did not exist until we ran commit on the 1st instance as well, since it was still in an uncommitted transaction. This has been explained in the previous page and the current tables look like the one from the 2nd instance, without the changes to state-name and with removed rows.	
3	SQL>SELECT * FROM store; NUMBER CITY STATE ========== ========== ====== 3 Los Angeles Calif 4 Atlanta Georg 5 San Francisco Calif 7 Oakland Calif 8 El Cerrito Calif 5 rows found	SQL>select * from store; NUMBER CITY STATE ========== ========== ====== 3 Los Angeles Calif 4 Atlanta Georg 5 San Francisco Calif 7 Oakland Calif 8 El Cerrito Calif 5 rows found

Thank you for this assignment!