Lecture Notes On Architecture of RDBMS, SQL

Sunnie S. Chung
Data Warehouse: A Multi-Tiered Architecture

Data Sources
- Other sources
- Operational DBs

Data Storage
- Extract Transform Load Refresh
- Metadata
- Monitor & Integrator
- Data Warehouse

OLAP Engine
- OLAP Server
- Serve
- Analysis Query Reports Data mining

Front-End Tools
- Data Marts
- Other sources
- Operational DBs

9/12/2017
SunnieChung_Lecture_Notes
Figure 2.3
Component modules of a DBMS and their interactions.

9/12/2017 SunnieChung_Lecture_Notes
DBMS Parallel Architecture

Message Subsystem

AMP

AMP

AMP

AMP

Channel

PE

GATEWAY

R3  R8  R11

R1  R6  R4

R7  R2  R22

R12  R9  R5

9/12/2017

SunnieChung_Lecture_Notes
DBMS Parallel Architecture

- PARSING & OPTIMISING ENGINE (PE)
  - SQL Parser & Optimizer
  - Query Step Dispatcher
  - Session Manager
  - Input Data Conversion

Message Subsystem

AMP
R3 R8 R11
R4
AMP
R1 R6 R4
R2 R7 R22
AMP
R12 R9 R5

9/12/2017
SunnieChung_Lecture_Notes
DBMS Parallel Architecture

Message Subsystem

- Software/Hardware Communications Network -- BYNET
- Enables Scalability
- Dual BYNETs for Fault Tolerance
DBMS Parallel Architecture

Message Subsystem

ACCESS MODULE PROCESS (AMP)
- Searching & Sorting
- Row Joins
- Aggregations
- Index Management
- Journaling & Rollback

ACCESS MODULE PROCESS (AMP)
- Searching & Sorting
- Row Joins
- Aggregations
- Index Management
- Journaling & Rollback

9/12/2017

SunnieChung_Lecture_Notes
Parallelization

- User request parallelized by optimizer
  - Single processing step sent to all vprocs simultaneously
  - Optimized to only affected vprocs as appropriate
  - Multiple steps launched concurrently for a single request
  - Multiple requests launched concurrently for multiple users
- Allows hundreds of in flight requests for thousands of concurrent users
Locking

- Access lock
  - Allows query while updates active
  - Might see partial transaction
- Read lock
  - Allows only other readers
- Write lock
  - No read locks allowed
  - Access locks can see it
- Exclusive lock
  - Keep Out!
PARSER and EXECUTION PLAN FLOW

Steps to Dispatcher

SQL Request

SYNTAX

RESOLVE

CHECK SECURITY

OPTIMIZE

STEP GEN

STEP PACKAGING

Cache Mgmt

V / M

9/12/2017 SunnieChung_Lecture_Notes
Resolver

- Retrieve dictionary information
  - From dictionary cache if possible
  - Annotate skeleton tree (database, table, column)
- Derive new conditions using transitivity
- Handle views and macros
- Handle derived tables
- Identify access requirements
- Report semantic errors
Resolver - Views Handling

- Analyze for complex views
- Materialize a complex view and represent it as a temp table in the query tree
- Applicable query conditions are pushed into the materialized view (temp table)
- Example:
  SQL:
  
  ```sql
  Create View v1 (x1, sum_y1) As
  Select x1, Sum(y1) From t1 Group By 1;
  Select * From t2, v1 Where x1=x2 and x1 > 10;
  ```

  At Resolver:
  
  ```sql
  ViewTable = Select x1, Sum(y1) From t1 Where x1>10 Group By1;
  ```

  Query “Sel * From t1, ViewTable Where x1=x2” is presented to Optimizer
Optimizer Input and Output

Resolver Tree

OPTIMIZER

Step list that specifies “best” query execution (join) plan
Example Query

(a) Query Q1

SELECT P.ins_class, Max(P.age)
FROM Patients P, Claims C
WHERE P.pid = C.pid AND P.age > 50
GROUP BY P.ins_class
HAVING COUNT (*) > 1;
Execution Semantics

```
SELECT P.ins_class, Max(P.age)
FROM Patients P, Claims C
WHERE P.pid = C.pid AND P.age > 50
GROUP BY P.ins_class
HAVING COUNT(*) > 1;
```

1. Construct a joined table of all the tables in the `FROM` list, and apply the predicates in `WHERE`.
2. Eliminate unwanted columns, Keep the columns listed in `SELECT, GROUP-BY`.
3. Sort the result by the columns in the `GROUP BY` clause to identify the groups.
4. Apply the predicates in the `HAVING` clause to each group.
5. Continued with more steps following…
Execution Semantics - Continued

SELECT P.ins_class, Max(P.age)  
FROM Patients P, Claims C  
WHERE P.pid = C.pid AND P.age > 50  
GROUP BY P.ins_class  
HAVING COUNT (*) > 1;

5. Generate the answer row per the remaining group columns generated by applying aggregate operators to each group.

6. Eliminate the duplicates if there is DISTINCT in the SELECT clause.
Execution Semantics

1. Construct a joined table of all the tables in the `FROM` list, and apply the predicates in `WHERE`.
2. Eliminate unwanted columns, Keep the columns listed in `SELECT, GROUP-BY`.
3. Sort the result by the columns in the `GROUP BY` clause to identify the groups.
4. Apply the predicates in the `HAVING` clause to each group.
5. Generate the answer row per the remaining group columns generated by applying aggregate operators to each group.
6. Eliminate the duplicates if there is `DISTINCT` in the `SELECT` clause.
Nested Queries: Non Correlated SubQuery

Q1: SELECT S.sname
    FROM Sailors S
    WHERE S.sid IN (SELECT R.sid
                     FROM Reserves R
                     WHERE R.bid=103);

- A SQL Query can itself contain another SQL Query in:
  - WHERE clause
  - FROM clause
  - HAVING clause

- Semantics of nested queries: Think of a nested loops evaluation:
  - Per tuple of outer table, evaluate subQ

- For each Sailors tuple,
  check the qualification by computing the subquery

- For non-correlated subQ: For Optimization, it is usually computed first once, saved as an intermediate table, then outer query is evaluated with it.
Nested Queries: Correlated SubQuery

Q2:

```
SELECT D.name
FROM Dept D
WHERE D.budget < 10000 and D.num_emps > (SELECT COUNT(*)
FROM Emp E
WHERE D.building = E.building);
```

- Subquery computation is per-row processing - NOT Possible to compute first for once
- Per tuple of outer table, evaluate subQ repeatedly
  - Bad when Outer Tab is large and a lot of Dup on Correlation Col
  - Good when Outer Tab is small with Index Lookup
- Scalar Aggregation:
  - Result always returns exactly one row for each outer row
  - For Empty input : 0 for Count, Null for Sum/Ave/Max/Min
Nested and Correlated Queries with Aggregation at Top level query

Q3:

```
SELECT P.ins_class, AVE(P.age) AS avgage
FROM Patients P
WHERE P.age ≥ 21
GROUP BY P.ins_class
HAVING 100 <
( SELECT COUNT(*)
FROM Patients P2
WHERE P2.age > 50 AND P.ins_class = P2.ins_class);
```
Query Execution Semantics of Q3

1. Compute Top level (outer) query
   1-1) From Patients, apply the predicate in WHERE, call the result Temp1.
   1-2) Project ins_class, pid, age, call the result Temp2.
   1-3) Sort the remaining tuples on ins_class, call the sorted table Temp3.
   1-4) For each group P.ins_class of Temp3 in Outer Q,
       Compute Sub query as follow:
       1-4-1) From Patients, Evaluate the predicates in the sub query, call it Temp4
       1-4-2) Compute COUNT(*) in Sub query.
       1-4-3) Apply the predicate in HAVING with the computed sub query result, call the result Temp5.

2. Scan Temp5 in the top level query maintaining the running info to compute AVE(P.age) in the top level query.

3. Output as in the SELECT clause in the top level query.
Example: Quantified Subqueries

```
Select p_partkey, p_name
From part
Where p_size >= x1 and p_size <= x2 and
  p_retailprice < all
  (Select ps_supplycost
   From partsupp
   Where ps_partkey = p_partkey and
     ps_availqty < y and
     not exists
       (Select *
        From lineitem
        Where p_partkey = l_partkey and
          ps_suppkey = l_suppkey and
          l_quantity = z)))
```
Query Tree: using Extended Nested Relational Algebra

\[ \pi_{p\text{-partkey}, p\text{-name}} \]
\[ \sigma_{p\text{-retailprice} < \text{ALL} \{ p\text{-supplycost} \} \lor \text{p\text{-partkey} is null}} \]
\[ \cup \{ p\text{-partkey}, p\text{-name}, p\text{-retailprice} \}, \{ p\text{-supplycost}, p\text{-partkey} \} \]
\[ \sigma' \text{ NOT EXISTS} \{ l\text{-orderkey} \} \]
\[ \cup \{ p\text{-partkey}, p\text{-name}, p\text{-retailprice}, p\text{-supplycost}, p\text{-partkey} \}, \{ l\text{-orderkey} \} \]
\[ \times_{p\text{-partkey} = l\text{-partkey}} \land p\text{-supplykey} = l\text{-supplykey} \]
\[ \times_{p\text{-partkey} = \text{ps\text{-partkey}}} \]
\[ \sigma_{l\text{-quantity} = z} \]
\[ \sigma_{p\text{-size} > x1} \land p\text{-size} \leq x2 \]
\[ \sigma_{\text{ps\text{-available} < y}} \]
lineitem

part

partsupp
Overview of Query Processing

1. SQL query
   - parse
   - parse tree
2. Query rewriting
3. Logical query plan
4. Physical plan generation
5. Physical query plan
6. Execute
   - result
Example Query

Select B,D
From R,S
Where R.A = "c" ∧ R.C=S.C
Example: Parse Tree

\[
\begin{align*}
\text{SELECT } & B, D \\
\text{FROM } & R, S \\
\text{WHERE } & R.A = "c" \land R.C = S.C
\end{align*}
\]
Along with Parsing ...

- Semantic checks
  - Do the projected attributes exist in the relations in the From clause?
  - Ambiguous attributes?
  - Type checking, ex: R.A > 17.5

- Expand views
Query rewriting

- SQL query
- Parse
- Parse tree
- Logical query plan
- Initial logical plan
- Logical plan
- Rewrite rules
- "Best" logical plan
- Physical plan generation
- Physical query plan
- Execute
- Result

Statistics
Initial Logical Plan

Relational Algebra: \( \Pi_{B,D} [ \sigma_{R.A = "c" \land R.C = S.C} (R \times S)] \)
Apply Rewrite Rule (1)

\[ \pi_{B,D} \left[ \sigma_{R.C=S.C} \left( \sigma_{R.A=\text{"c"}} (R \times S) \right) \right] \]
Apply Rewrite Rule (2)

\[ \pi_{B,D} \left[ \sigma_{R.C = S.C} \left[ \sigma_{R.A = "c"}(R) \right] \times S \right] \]
Apply Rewrite Rule (3)

\[ \pi_{B,D} \sigma_{R.A = "c"} (R) \]

\[ \sigma_{R.C = S.C} \]

\[ \pi_{B,D} \sigma_{R.A = "c"}(R) \bowtie S \]

\[ \Pi_{B,D} [[\sigma_{R.A = "c"}(R)] \bowtie S] \]
The diagram illustrates the process of query execution in a database system.

1. **SQL query** is parsed to create a **parse tree**.
2. **Query rewriting** takes place, involving **rewrite rules** to transform the query.
3. **Statistics** are collected to generate a **logical query plan**.
4. The logical plan is optimized to find the **"Best" logical plan**.
5. A **physical query plan** is generated from the optimized logical plan.
6. The query is **executed** using the physical plan.
7. The **result** is produced.

The process is iterative, with the ability to refine the logical plan to generate a more efficient physical plan.
Physical Query Execution Plan

Select E.ssn
From   Works_On W, Employee E
Where W.essn = E.ssn
Group By E.ssn
Having Count(E.ssn) > 2;
Physical Query Execution Plan
GRANT

- SQL GRANT is a command used to provide access or privileges on the database objects to the users.
- The Syntax for the GRANT command is:
  ```
  GRANT privilege_name
  ON object_name
  TO {user_name | PUBLIC | role_name}
  [WITH GRANT OPTION];
  ```
  - `privilege_name` is the access right or privilege granted to the user. Some of the access rights are ALL, CREATE TABLE, DROP TABLE, EXECUTE, and SELECT, INSERT/DELETE/UPDATE.
  - `object_name` is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
  - `user_name` is the name of the user to whom an access right is being granted.
  - `PUBLIC` is used to grant access rights to all users.
  - `ROLES` are a set of privileges grouped together.
  - `WITH GRANT OPTION` - allows a user to grant access rights to other users.
  - Example:
    ```
    GRANT Create Table ON Employee TO PUBLIC WITH GRANT OPTION;
    ```
REVOKE

- The REVOKE command removes user access rights or privileges to the database objects.
- The Syntax:
  ```sql
  REVOKE privilege_name
  ON object_name
  FROM {user_name |PUBLIC |role_name};
  ```
- Example:
  ```sql
  REVOKE Create Table ON Employee FROM Public;
  ```
ACID

- **Atomicity**
  Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

- **Consistency**
  The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

- **Isolation**
  The isolation property ensures that the concurrent execution of transactions result in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.

- **Durability**
  Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.
Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables
  - Considered to be a virtual table
Specification of Views in SQL

- **CREATE VIEW** command
  - Give table name, list of attribute names, and a query to specify the contents of the view

V1: CREATE VIEW WORKS_ON1 
    AS SELECT Fname, Lname, Pname, Hours 
    FROM EMPLOYEE, PROJECT, WORKS_ON 
    WHERE Ssn=Esn AND Pno=Pnumber;

V2: CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal) 
    AS SELECT Dname, COUNT (*), SUM (Salary) 
    FROM DEPARTMENT, EMPLOYEE 
    WHERE Dnumber=Dno 
    GROUP BY Dname;
Specification of Views in SQL (cont’d.)

- Specify SQL queries on a view: Select Queries only (No DML)
- View always up-to-date
  - Responsibility of the DBMS and not the User
- **DROP VIEW** command
  - Dispose of a view
Levels of Abstraction

External view; user and data designer

Logical storage; data designer

Physical storage; DBA
Physical Schema

The physical schema is a description of how the data is physically stored in the database. It includes:

- Where the data is located
- File structures
- Access methods
- Indexes

The physical schema is managed by the DBA.
Conceptual Schema

The conceptual schema is a logical description of how the data is stored. It consists of the schemas we have described with CREATE TABLE statements. It is managed by the data designer.
External Schemas

Each external schema is a combination of base tables and views, tailored to the needs of a single user. It is managed by the data designer and the user.
Data Independence

- A database model possesses *data independence* if application programs are immune to changes in the conceptual and physical schemas.
- Why is this important? Everything changes.
- How does the relational model achieve logical (conceptual) data independence?
  - Through views
  - If the conceptual schema changes, a view can be defined to preserve existing applications.
Data Independence (ctd.)

- How does the relational model achieve physical data independence?
  - Conceptual level contains no physical info
  - SQL can program against the conceptual level

Earlier DBMSs (network, hierarchical) did not have these properties.
Their languages had physical properties embedded in them.

- That is the primary reason for the success of the relational model
Views

Students(sid, name, address, gpa)
Completed( sid, course, grade)

- A view is a query stored in the database
  - Think of it as a table definition for future use
- Example view definition:
  CREATE VIEW gstudents AS
  SELECT *
  FROM Students
  WHERE gpa >= 2.5

- Views can be used like base tables, in any query or in any other view. Like a Macro.
Example view use: simpler queries

- Suppose you want to retrieve good students who have completed CIS530.
  
  ```sql
  SELECT S.name, S.phone
  FROM gstudents S JOIN completed C
  ON S.sid = C.sid
  WHERE C.course = 'CIS530';
  ```

- It’s easier to write the query using the view.
Views for Security

- This is the student table without the gpa field to hide gpa to any user in DB.

CREATE VIEW sstudents AS
SELECT sid, name, address
FROM students
Views for Extensibility

- An old company’s database includes a table: `Part (PartID, Name, Weight)`
- Weight is stored in pounds
- The company is purchased by a new firm that uses metric weights
- The two databases, old and new, must be integrated and use Kg.
- But there’s lots of old software using pounds.
- Solution: views!
Views for extensibility (ctd)

- Solution:
  1. Base table with kilograms becomes NewPart, for new integrated company.
  2. `CREATE VIEW Part AS
     SELECT PartID, Name, 
     2.2046*Weight
     FROM NewPart;
  3. Old programs still call the table “Part”
CREATE VIEW PartitionedView AS
    SELECT *
    FROM MyDatabase.dbo_PartitionTable1
UNION ALL
    SELECT * FROM Server2.MyDatabase.dbo_PartitionTable2
UNION ALL
    SELECT * FROM Server3.MyDatabase.dbo_PartitionTable3
View Update and Inline Views

- Clause **WITH CHECK OPTION**
  - Must be added at the end of the view definition if a view is to be updated

- In-line view
  - Defined in the *FROM* clause of an SQL query

```
Select D.dnumber, Count(*)
From Employee E, Department D,
    (Select E.dno
     From Employee E
     Group By E.dno
     Having COUNT(*) > 2) as Temp(high_dno)
Where  D.dnumber = E.dno and  Temp.high_dno = e.dno
Group by D.dnumber;
```
View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Query modification** approach
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute
View Processing

- View is a **virtual** table
- **How a view is defined:**
  
  CREATE VIEW **V_ATL-FLT**
  AS SELECT FLT#, AIRLINE, PRICE
  FROM FLT-SCHEDULE
  WHERE FROM-AIRPORTCODE = “ATL”;
- How a query on a view is written:
  
  SELECT *
  FROM **V_ATL-FLT**
  WHERE PRICE <= 00200.00;
- How a query on a view is computed:
  
  SELECT FLT#, AIRLINE, PRICE
  FROM FLT-SCHEDULE
  WHERE FROM-AIRPORTCODE = “ATL”
  AND PRICE < 00200.00;
- How a view definition is dropped:
  
  DROP VIEW ATL-FLT [RESTRICT|CASCADE];
View Implementation

- View materialization approach
  - Physically create a temporary view table when the view is first queried
  - Keep that table on the assumption that other queries on the view will follow
  - Requires efficient strategy for automatically updating the view table when the base tables are updated
View Implementation (cont’d.)

- **Incremental update strategies**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table
View Update

- When View is replaced with the definition (underlying base tables), optimizer builds an execution plan: View will be executed over updated base tables so View is always updated. – This is the most general case
- When the view is very complex and expected to be used again, the execution plan is cached: when the saved execution plan is picked up again, it may not always updated with schema changes in base tables. Some DBMS delay the propagation of schema changes to the view until the view is altered/replaced. Need to refresh view:
  EXEC sp_refreshview @ViewName;
- However, Sql Server updates the cached plan when indexes of base tables are changed
- When the view is very complex and frequently used, it is often materialized: Sometimes DBMS may incrementally update the view to the changes of the base tables. Need to refresh view
  EXEC sp_refreshview @ViewName;
View Update Tips

- You should run:
  - --First option
    EXEC sp_refreshview @ViewName
  - --Second option
    CREATE PROCEDURE REFRESH_ALL_VIEWS
      AS
      DECLARE @ViewName varchar(100)
      DECLARE curViews CURSOR FOR select name from sysobjects where xtype='V'
      OPEN curViews
      FETCH NEXT FROM curViews INTO @ViewName
      WHILE @@FETCH_STATUS = 0
        BEGIN
          EXEC sp_refreshview @ViewName
          FETCH NEXT FROM curViews INTO @ViewName
        END
      CLOSE curViews
      DEALLOCATE curViews
    
    GO
The data "in" a view has no existence independent from the tables that make up the view. The view is, in essence, a stored SELECT statement that masquerades as a table. The data is stored in the original tables and only "assembled" into the view when you want to look at it. If the view is updateable (not all views are) the updates are applied to the table data.

In general, there are no indexes on views, only on the underlying tables. The indexes are used by the DBMS to assemble the rows from the different tables when you request data from the view. (This ignores the "materialized view" in which the database stores a "shadow" copy of the data, effectively pre-computing the view for you).

However, Indexed view is possible in limited cases.
Problem with views: update

- Views cannot always be updated unambiguously. Consider

\[ \text{Emp}(\text{empid, ename, address, deptid}) \]
\[ \text{Dept}(\text{deptid, dname}) \]

\[
\text{CREATE VIEW EMPDEPT AS}
\text{SELECT ename, dname}
\text{FROM Emp JOIN Dept USING (deptid)}
\]

<table>
<thead>
<tr>
<th>EMPDEPT</th>
<th>ename</th>
<th>dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>shoe</td>
<td></td>
</tr>
<tr>
<td>joe</td>
<td>suit</td>
<td></td>
</tr>
</tbody>
</table>

- I want to delete (jim, shoe) from EMPDEPT.
- Can I do that? No!
The good news

A view can be updated if

- It is defined on a single base table
- Using only selection and projection
- No aggregates
- No DISTINCT
Limits on Updating View

this MSDN article: Modifying Data Through a View,
Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table.

The columns that are being modified in the view must reference the underlying data in the table columns directly. They cannot be derived in any other way, such as through:

- An aggregate function (AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR and VARP).
- A computation; the column cannot be computed from an expression using other columns. Columns formed using set operators (UNION, UNION ALL, CROSSJOIN, EXCEPT, and INTERSECT) amount to a computation and are also not updatable.

The columns that are being modified cannot be affected by GROUP BY, HAVING, or DISTINCT clauses.

TOP cannot be used anywhere in the select_statement of the view when WITH CHECK OPTION is also specified.
Updatable View and Inline Views

- Update on a view defined on a **single table** without any aggregate functions
  - Can be mapped to an update on underlying base table (only on single table)
  - `DELETE FROM my_View WHERE id = 3;` when `my_View` is defined on single table

- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended
Updatable View

The example inserts a new row into the base table HumanResources.Department by specifying the relevant columns from the view HumanResources.vEmployeeDepartmentHistory.

The statement succeeds because only columns from a single base table are specified and the other columns in the base table have default values.

USE AdventureWorks2012;
INSERT INTO HumanResources.vEmployeeDepartmentHistory (Department, GroupName) VALUES ('MyDepartment', 'MyGroup');
Updatable View

CREATE VIEW titleview AS
    SELECT title, au_ord, au_lname, price, ytd_sales, pub_id
    FROM authors AS a JOIN titleauthor AS ta ON (a.au_id = ta.au_id)
    JOIN titles AS t ON (t.title_id = ta.title_id);

SELECT * FROM titleview;

CREATE VIEW Cust_titleview AS
    SELECT title, au_lname, price, pub_id FROM titleview;

Views in all versions of SQL Server are updatable (can be the target of UPDATE, DELETE, or INSERT statements), as long as the modification affects only one of the base tables referenced by the view, for example:

-- Increase the prices for publisher '0736' by 10%.
UPDATE titleview SET price = price * 1.10 WHERE pub_id = '0736';
# Views vs Tables

<table>
<thead>
<tr>
<th>Creating</th>
<th>Create view $V$ as (select * from $A$, $B$ where ...)</th>
<th>Create table $T$ as (select * from $A$, $B$ where ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used</td>
<td>In any select query. Only some update queries.</td>
<td>It’s a new table. You can do what you want.</td>
</tr>
<tr>
<td>Maintained as</td>
<td>1. Evaluate the query and store it on disk as if a table. 2. Don’t store. Replace it with the query when referenced.</td>
<td>It’s a new table. Stored on disk.</td>
</tr>
<tr>
<td>What if a tuple inserted in $A$?</td>
<td>1. If $V$ is stored on disk, the stored table is automatically updated to be accurate. 2. If we are just replacing $V$ with the query, there is no need to do anything.</td>
<td>$T$ is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger to update $T$ whenever $A$ is updated.</td>
</tr>
</tbody>
</table>
Transaction with Commit

- Use the COMMIT statement to end your current transaction and make permanent all changes performed in the transaction.
- A transaction is a sequence of SQL statements that a Database treats as a single unit.
- This statement also erases all savepoints in the transaction and releases transaction locks.
COMMIT

- Until you commit a transaction:
  - You can see any changes you have made during the transaction by querying the modified tables, but other users cannot see the changes.
  - After you commit the transaction, the changes are visible to other users' statements that execute after the commit.
  - You can roll back (undo) any changes made during the transaction with the ROLLBACK statement.
COMMIT

Example:
USE AdventureWorks2012;
GO
BEGIN TRANSACTION;
GO
DELETE FROM HumanResources.JobCandidate
WHERE JobCandidateID = 13;
GO
COMMIT TRANSACTION;
GO
RollBack

- The ROLLBACK statement is the inverse of the COMMIT statement.
- It undoes some or all database changes made during the current transaction.
COMMIT/ROLLBACK

- Issuing a COMMIT TRANSACTION when @@TRANCOUNT is 0 results in an error; there is no corresponding BEGIN TRANSACTION.
- You cannot roll back a transaction after a COMMIT TRANSACTION statement is issued because the data modifications have been made a permanent part of the database.
Rollback with Transaction

- USE tempdb;
- GO
- CREATE TABLE ValueTable ([value] int);
- GO
- DECLARE @TransactionName varchar(20) = 'Transaction1';
- --The following statements start a named transaction, --insert two rows, and then roll back --the transaction named in the variable @TransactionName.
- --Another statement outside of the named transaction inserts two rows.
- --The query returns the results of the previous statements.
- BEGIN TRAN @TransactionName
- INSERT INTO ValueTable VALUES(1), (2);
- ROLLBACK TRAN @TransactionName;
- INSERT INTO ValueTable VALUES(3),(4);
- SELECT [value] FROM ValueTable;
- DROP TABLE ValueTable;

- --Results --value
- -------------- --3
- --4
Rollback with SavePoint

- SAVEPOINT names and marks the current point in the processing of a transaction.
- Savepoints let you roll back part of a transaction instead of the whole transaction.
- The number of active savepoints for each session is unlimited.
- Roll back to the savepoint below, undoing just the insert.

```sql
CREATE TABLE emp_name AS SELECT employee_id, last_name, salary FROM employees;
CREATE UNIQUE INDEX empname_ix ON emp_name (employee_id);
DECLARE emp_id employees.employee_id%TYPE;
emp_lastname employees.last_name%TYPE;
emp_salary employees.salary%TYPE;
BEGIN
  SELECT employee_id, last_name, salary INTO emp_id, emp_lastname, emp_salary
  FROM employees WHERE employee_id = 120;
  UPDATE emp_name SET salary = salary * 1.1 WHERE employee_id = emp_id;
  DELETE FROM emp_name WHERE employee_id = 130;
  SAVEPOINT do_insert;
  INSERT INTO emp_name VALUES (emp_id, emp_lastname, emp_salary);
EXCEPTION WHEN DUP_VAL_ON_INDEX THEN
  ROLLBACK TO do_insert;
  DBMS_OUTPUT.PUT_LINE('Insert has been rolled back');
END;
```
ROLLBACK with SAVEPOINT

- Savepoint is not a variable. It only has the most recent savepoint even if it is used again in the same transaction.
- Duplicate savepoint names are allowed in a transaction, but a ROLLBACK TRANSACTION statement that specifies the savepoint name will only roll the transaction back to the most recent SAVE TRANSACTION using that name.
ROLLBACK with Error

A ROLLBACK TRANSACTION statement does not produce any messages to the user.

Error handling statement with ROLL Back

If warnings are needed in stored procedures or triggers, use the RAISERROR or PRINT statements. RAISERROR is the preferred statement for indicating errors.
BEGIN TRANSACTION;
BEGIN TRY  -- Generate a constraint violation error.
    DELETE FROM Production.Product WHERE ProductID = 980;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber, ERROR_SEVERITY() AS ErrorSeverity
    , ERROR_STATE() AS ErrorState, ERROR_PROCEDURE() AS ErrorProcedure
    , ERROR_LINE() AS ErrorLine, ERROR_MESSAGE() AS ErrorMessage;
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
END CATCH;
END TRANSACTION;
IF @@TRANCOUNT > 0 COMMIT TRANSACTION;
GO
If an Error occurs in a transaction, the entire transaction will be rolled back. You should issue the command to roll it back.

You can wrap this in a TRY CATCH block as follows

BEGIN TRY
    BEGIN TRANSACTION
    INSERT INTO myTable (myColumns ...) VALUES (myValues ...);
    INSERT INTO myTable (myColumns ...) VALUES (myValues ...);
    INSERT INTO myTable (myColumns ...) VALUES (myValues ...);
    COMMIT TRAN -- Transaction Success!
END TRY

BEGIN CATCH
    IF @@TRANCOUNT > 0
        ROLLBACK TRAN -- RollBack in case of Error
    -- you can Raise ERROR with RAISEERROR() Statement including the details of the exception
    RAISERROR(ERROR_MESSAGE(), ERROR_SEVERITY(), 1)
END CATCH
SET TRANSACTION ISOLATION LEVEL

SET TRANSACTION ISOLATION LEVEL

{ READ UNCOMMITTED
  | READ COMMITTED
  | REPEATABLE READ
  | SNAPSHOT
  | SERIALIZABLE  };

SET TRANSACTION ISOLATION LEVEL:

SET TRANSACTION ISOLATION LEVEL

{  
  | READ UNCOMMITTED: Dirty Read. Can read the ones under being modified but not committed by other transactions
  | READ COMMITTED: Can’t read the ones not committed by other transaction. Other transaction can modify between statements by this transaction. Default. Non repeatable read result in phantom read
  | REPEATABLE READ: can’t read the ones under being modified and not committed by other transaction and other transactions can not modify until the current transaction is completed.
  | SNAPSHOT:
  | SERIALIZABLE:
};
SET TRANSACTION ISOLATION LEVEL:

SET TRANSACTION ISOLATION LEVEL

{ READ UNCOMMITTED:
| READ COMMITTED:
| READ | REPEATABLE READ:
| SNAPSHOT: Data Version control. Any transaction has consistent data before the transaction begins. For file streaming data.
| SERIALIZABLE: can’t read the ones under being modified and not committed by other transaction and other transactions can not modify until the current transaction is completed. Other transaction can’t insert new data with the key value that falls in the range read by any statement in the current transaction };