Embedded SQL

- Use SQL from inside a programming language
- Any language: C++, Java, etc
- We look at general, non-language specific aspects
- Why use embedded SQL?
  - Formatting (e.g. HTML)
  - Express queries not expressible in SQL (complex recursion)
  - Database integration, extensions etc.

Basic idea

- SQL statements used as subprocedures
- Main problem: Arguments and results in C++ (for example) have types
- Basic type or relation doesn't exist in C++, and SQL doesn't have pointers etc.

Basic concepts

- `EXEC SQL`
- Followed by SQL statement or statements (surrounded by `BEGIN` and `END`
- Shared variables. Prefix ‘: in SQL, without ‘:’ in C++
- Special variable `SQLSTATE` (array of 5 characters)

Variable declarations

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAddr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

Declares variables that can be used both in C++ program and in SQL calls

We describe a function that asks user for values of attributes and inserts them into `Studio` (we omit non-SQL parts of the code)

Insertion example

```
void getStudio() {

EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAddr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

/* print request that studio name and address be entered and read
response into variables studioName and studioAddr */

EXEC SQL INSERT INTO Studio(name, address)
    VALUES (:studioName, :studioAddr);
}
```

This function has no output, so integration with SQL is easy

Queries with output

- Single-tuple `SELECT`: Direct communication with shared variables
- General queries: *cursors*
- Example: Read studio name, and print president's `netWorth`

Example

```
void printNetWorth {

   EXEC SQL BEGIN DECLARE SECTION;
      charstudioName [50]; int presNetWorth;
      char SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;

   /* print request that studio name be entered, read response into
      studioName */

   EXEC SQL SELECT netWorth
            INTO :presNetWorth
            FROM Studio, MovieExec
            WHERE presC# = cert# AND
                  Studio.name = :studioName;

   /* check that SQLSTATE has all 0s (meaning: no error) and if so,
      print the value of presNetWorth */

}
```

Cursors

- Cursor: A pointer that iterates over tuple in a relation
- Relation can be base relation or result of query
- Declare cursor

  `EXEC SQL DECLARE <cursor name> CURSOR FOR <query>`

  `<query>` is relation name or SQL query
- Initialize cursor

  `EXEC SQL OPEN <cursor name>`

  Cursor is now ready to get the first tuple
- Opening is separate from declaration so that we can examine the relation more than once

Fetching tuples

- Gets the next tuple of the relation over which the cursor ranges
- Syntax:

  `EXEC SQL FETCH FROM<cursor name> INTO <list of variables>`

- One variable in the list for each attribute of the relation
- Reads the tuple and advances to the next tuple (even if it doesn't exist)
- If we are at the last tuple, return nothing and set `SQLSTATE` to `02000` (no need to memorize these codes...)

```
void worthRanges() {

   int i, digits, counts[15];
   EXEC SQL BEGIN DECLARE SECTION;
      int worth;
      char SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;
   EXEC SQL DECLARE execCursor CURSOR FOR
      SELECT netWorth FROM MovieExec;

   EXEC SQL OPEN execCursor;
   for (i=1; i<15; i++) counts[i] = 0;
   while(1) {
      EXEC SQL FETCH FROM execCursor INTO :worth;
   if (NO_MORE_TUPLES) break;
   digits = 1;
   while((worth /= 10) > 0)  digits++;
   if(digits <= 14) counts[digits]++;
   }
   EXEC SQL CLOSE execCursor;
   for(i=0; i<15; i++)
      printf ("digits = %/d: number of execs = %d\n",
         i, counts[i]);
}
```

Modifcations

- We can modify the tuple at the current location of the cursor
- Syntax is standard SQL with the condition `WHERE CURRENT`
- Example: examine value of `netWorth`:
  - If it is less than 1.000, delete the tuple
  - Otherwise, double the value

```
void changeWorth() {

   EXEC SQL BEGIN DECLARE SECTION;
      int certNo, worth;
      char execName [31], execAddr[256], SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;
   EXEC SQL DECLARE execCursor CURSOR FOR MovieExec;

   EXEC SQL OPEN execCursor;
   while(1) {
      EXEC SQL FETCH FROM execCursor INTO :execName,
         :execAddr, :certNo, :worth;
      if (NO_MORE_TUPLES) break;
      if (worth < 1000)
         EXEC SQL DELETE FROM MovieExec
            WHERE CURRENT OF execCursor;
      else
         EXEC SQL UPDATE MovieExec
            SET netWorth = 2 * netWorth
            WHERE CURRENT OF execCursor;
   }
   EXEC SQL CLOSE execCursor;
}
```

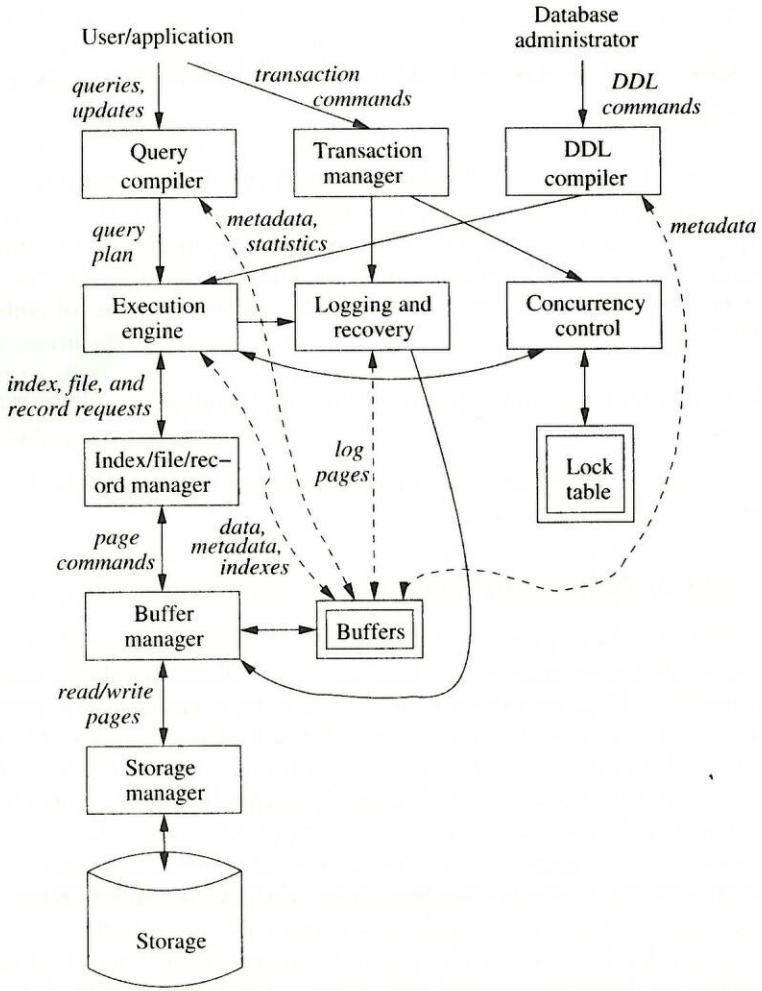
Dynamic SQL

- So far, the SQL statement is part of the C++ program
- What if we what to change the SQL query?
- Basic idea: Use a string variable to contain the SQL statement
- 'SQL variable": variable that holds an interpreted SQL string
- Then use special primitives to convert the string to SQL and execute
- Two statements
  - `EXEC SQL PREPARE V FROM <expression>` Analyse the SQL statement and find a good strategy to evaluate
  - `EXEC SQL EXECUTE V`: Execute the query
- We can prepare a query, and then evaluate it multiple times
- Can be combined in one statement `EXEC SQL EXECUTE IMMEDIATE <expression>`

Example: Read query from user input and execute it

```
void readQuery() {

   EXEC SQL BEGIN DECLARE SECTION;
      char *query;
   EXEC SQL END DECLARE SECTION;

   /* prompt user for a query, allocate space (e.g.,
      use malloc) and make shared variable :query point
      to the first character of the query */
   EXEC SQL PREPARE SQLquery FROM :query;
   EXEC SQL EXECUTE SQLquery;
}
```

SQL implementation



Query evaluation

- Parsing. Convert query into a parse tree
- Convert parse tree into an algebraic expression
- Algebraic optimization
- Physical plan generation

Optimization:

- Best algebraic expression for query evaluation
- Chose among different algorithms for each algebraic operator
- Decide how to pass data between the operations

Data storage

- Data in databases is usually stored on disks, and the cost of disk access is usually the dominant cost
- The most important thing to optimize is the number of disk accesses: only read the part of the database that we need, and try to avoid reading the data more than once, when possible
- We assume a fixed size of disk blocks, and assume that we can only read data in blocks (rather than tuples)

Indexes

- Index: A permanent data structure that lets us find efficiently all the tuples with a particular value, or range of values, for a given attribute or attributes (such as a key)
- The standard data structure, called a *B-tree* is a search tree as you have seen in your data structure course (depth $\log n$)
- There are two ways in which the B-tree differs from those that you have seen:
  - The leaves contain not the data itself but pointers to the blocks containing the data
  - It must be possible to *update* the tree efficiently, and with the B-tree this can be done in average time $\log n$ (technically amortized time, which is somewhat better)

Clustered index

Two types of index

- Clustered: All tuples with a given value for the index attribute are found on the same page (or adjacent one)
- Unclustered
- (Note: The book also discusses *clustering indexes* which are something different)

Important points

- If there are a small number of tuples (e.g. 10) with a given value for the index, we have to retrieve only 1 (or 2) blocks with a clustered index
- We may have to retrieve 10 blocks with an unclustered one
- We can have at most one clustered index per relation
- There is no limit on the number of unclustered indexes, but the cost in space and updating costs must be taken into account
- Range queries (attribute between k_1 and k_2) can also be answered with B-trees (with similar tradeoffs between clustered and unclustered indexes)

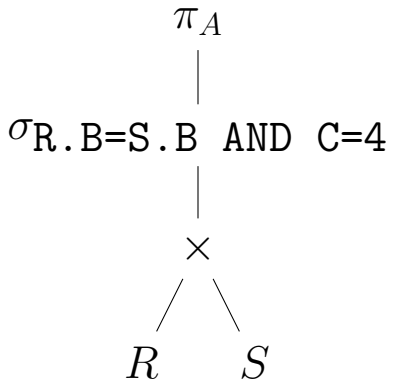
Parsing

- Parser analyses the query and converts it into a tree
- Checks for syntax errors
- Resolves ambiguity. Determines which relation each attribute belongs to
- Checks types: arithmetic only on numeric attributes etc.
- Converts view references to parse trees of their definitions
- We do not study this further. The book contains examples of a *grammar* for SQL. Constructing a parser is a standard application of techniques studied in formal languages

Conversion to relational algebra

- We start with two simple examples, one using a view
- We then discuss some aspects of nested queries
- First example: relations `R(A,B)`, `S(B,C)`

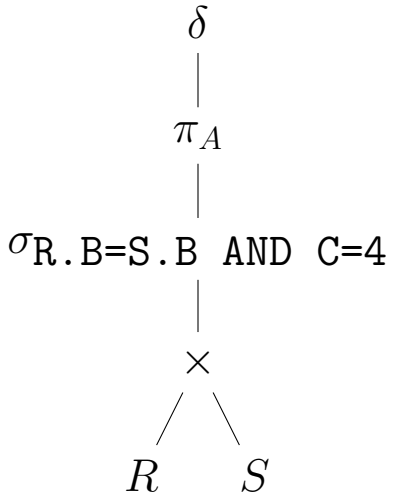
```
SELECT A
FROM R, S
WHERE R.B=S.B AND C=4
```



- FROM: Cartesian product
- WHERE: Selection
- SELECT: Projection

Another example:

```
SELECT DISTINCT A
FROM R, S
WHERE R.B=S.B AND C=4
```



Query with view

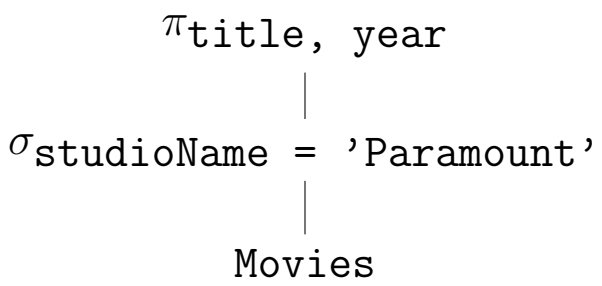
View

```
CREATE VIEW ParamountMovies AS
 SELECT title, year
 FROM Movie
 WHERE studioName = 'Paramount';
```

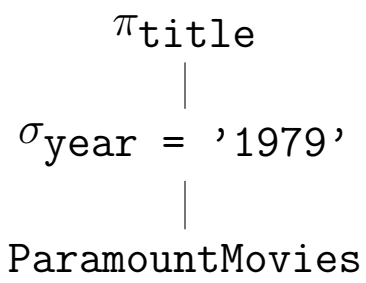
Query

```
SELECT title
FROM Paramount Movies
WHERE year = '1979';
```

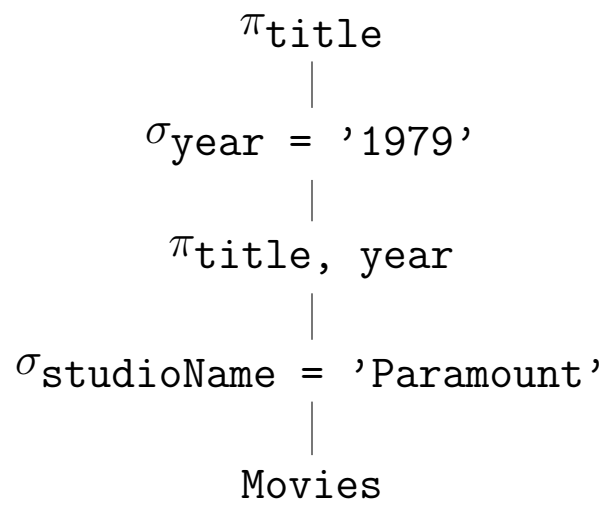
Algebraic expression for view



Algebraic expression for query



Combined



Note that the second projection isn't really needed, and should be removed later

Nested queries

- Correlated query: Subquery can refer to attibutes in the main query
- Hardest part of the algorithm
- We assume queries are not correlated, so we can convert the subqueries separately from the main query

Queries with `EXISTS`

```
SELECT A
FROM R
WHERE EXISTS Q
```

- Convert Q to algebraic expression $\hat{Q}$
- $R \times \hat{Q}$ is empty when `EXISTS Q` is false
- If `EXISTS Q` is true, $R \times \hat{Q}$ contains all the tuples in R used in the query (care needed with duplicates)
- We get $\pi_A(R \times Q)$
- `NOT EXISTS` is similar but a bit more complicated

Other subqueries

- Try to convert to queries with `EXISTS` or `NOT EXISTS`
- We don't give a formal description, but illustrate with several examples

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate = 1960)
```

Becomes

```
SELECT movieTitle FROM StarsIn
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate = 1960 AND name = starName)
```

Another example

```
SELECT name FROM MovieExec
WHERE netWorth >= ALL (SELECT E.netWorth
                       FROM MovieExec E)
```

Becomes

```
SELECT name FROM MovieExec
WHERE NOT EXISTS (SELECT E.netWorth
                  FROM MovieExec E
                  WHERE netWorth < E.netWorth)
```

Third example

```
SELECT C FROM S
WHERE C IN (SELECT SUM(B) FROM R
GROUP BY A)
```

Becomes

```
SELECT C FROM S
WHERE EXISTS (SELECT SUM(B) FROM R
              GROUP BY A
              HAVING SUM(B) = C)
```

Other primitives

- `GROUP BY`, `HAVING`: use γ
- `ORDER BY`: Use τ
- `UNION`, `INTERSECT`, `EXCEPT`: Use $\cup$, $\cap$, $-$

`SELECT` conditions

- `AND`: combine conditions
- `OR`: Use union (careful with duplicates)
- `NOT`: Use difference

Algebraic optimization

"Pushing" selections

- Selection usually reduces significantly the size of the relations
- Extreme example (but also typical): Selecting for a value of Codice Fiscale reduces a huge relation to a single tuple
- For this reason, we should do selection as soon as possible
- We use algebraic rules that we studied earlier
- For example:

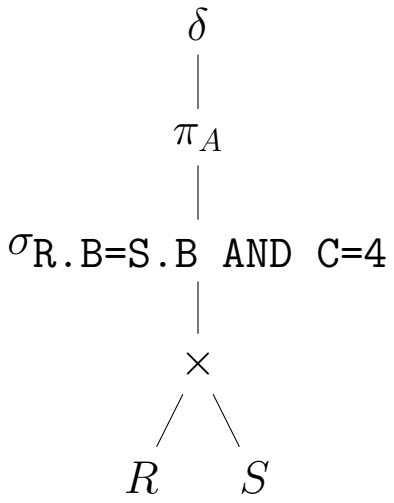
$$\sigma_C(\delta(R)) = \delta(\sigma_C(R))$$

or

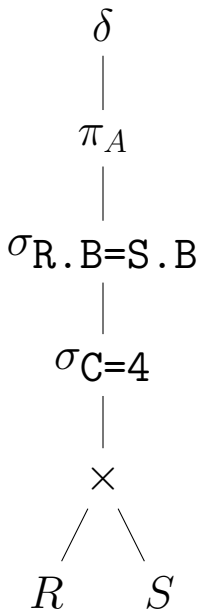
$$\sigma_C(R \times S) = \sigma_C(R) \times S$$

provided that C only uses attributes of R

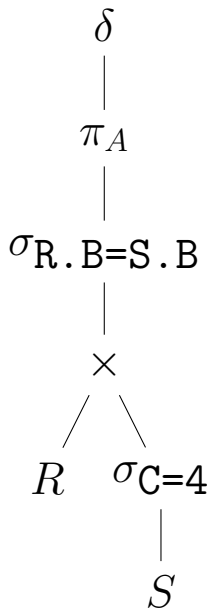
Example



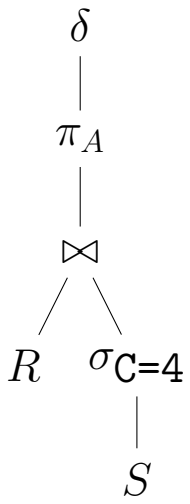
Is first converted to



We can then push the second selection down the right (not the left) branch to get the query plan



- Query interpreters have very efficient algorithms for join
- Therefore, replace cartesian product with selection by join when possible

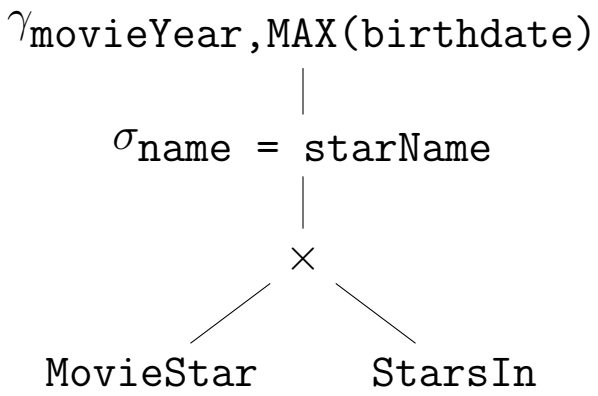


- Projections can sometimes be pushed down as well
- The saving is smaller and we do not study this aspect further
- In our example, the projection cannot be pushed down any further (why?)

Another example

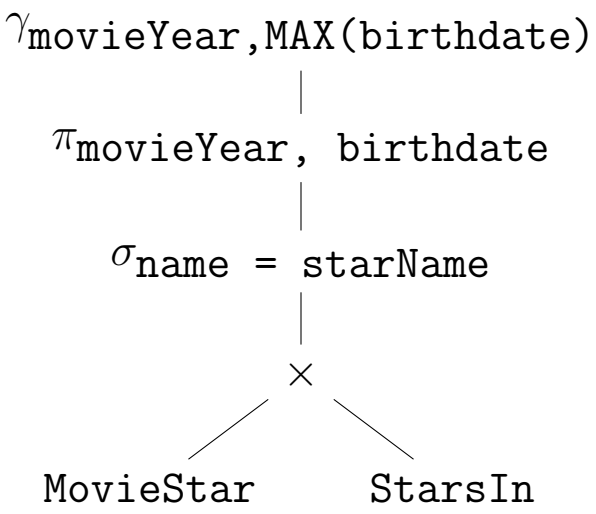
```
SELECT movieYear, MAX(birthdate)
FROM MovieStar, Starsln
WHERE name = starName
GROUP BY movieYear;
```

Becomes

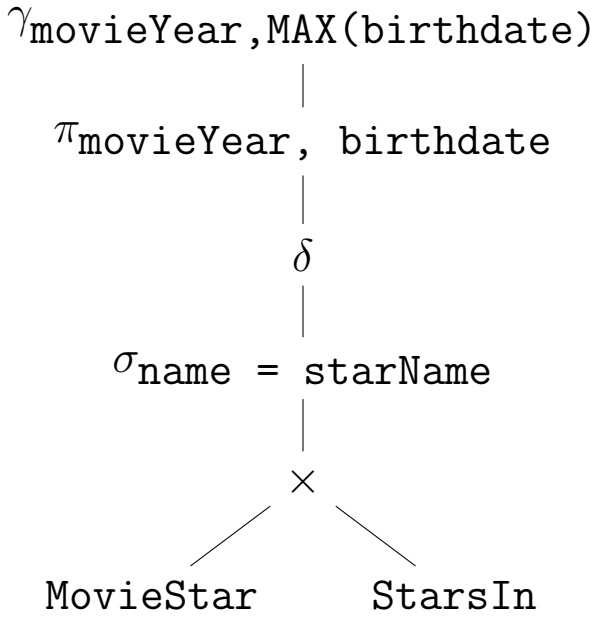


Possible transformations

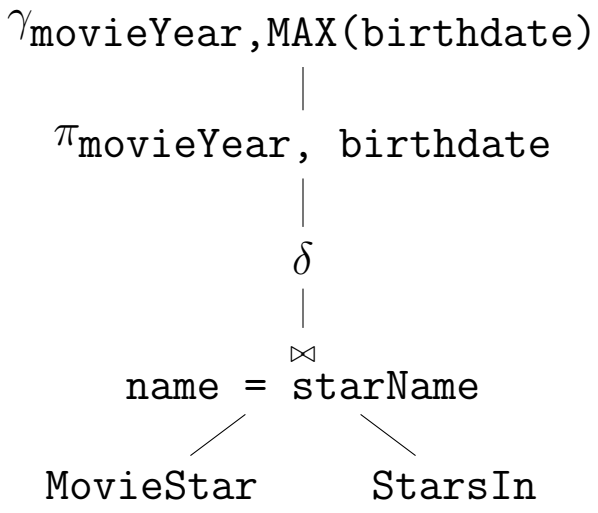
Introduce projection before γ, to remove unneeded attributes



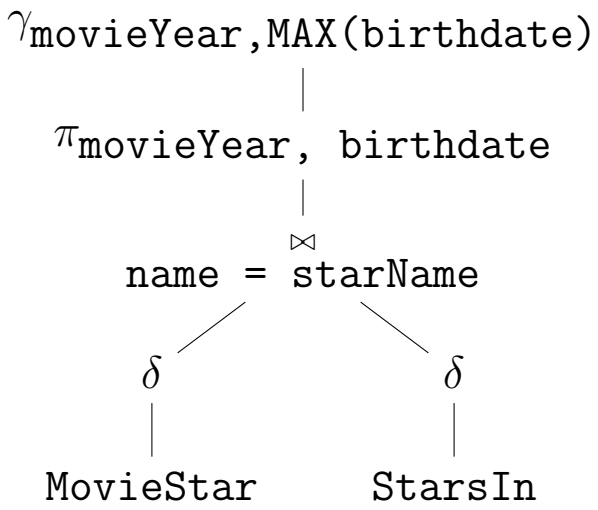
Introduce δ as γ (with MAX) is insensitive to duplicate elimination



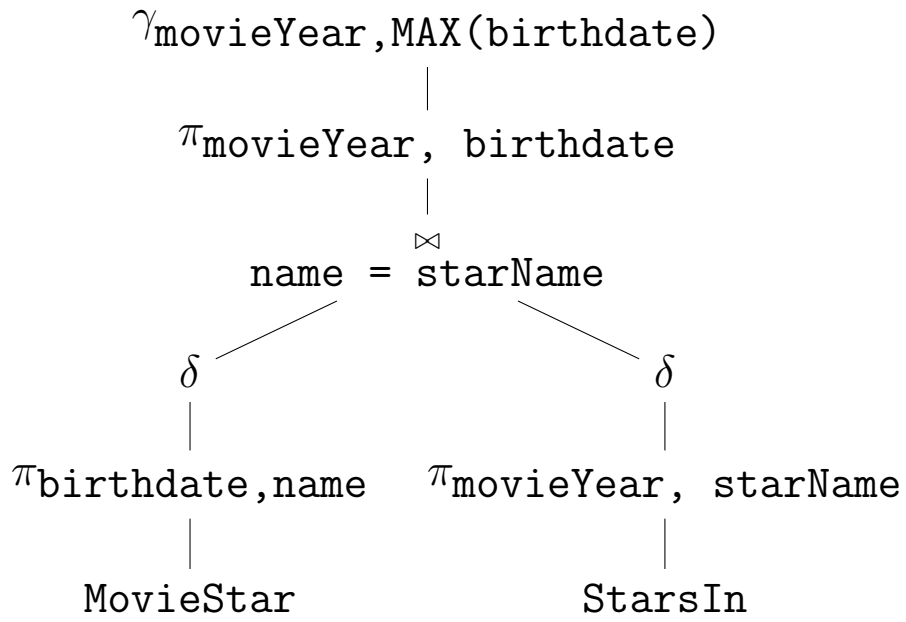
Combine selection and cartesian product into join



Push down δ



Add projections



Which of these algebraic plans are better?

- In general, there is no obvious answer
- The relative efficiency could depend on the data
- We shall study ways of using statistical properties of the data to determine the better plan
- These techniques are not perfect, and will sometimes give a wrong answer
- But errors affect the efficiency only, and not the corectness of the result

A few rules are always better

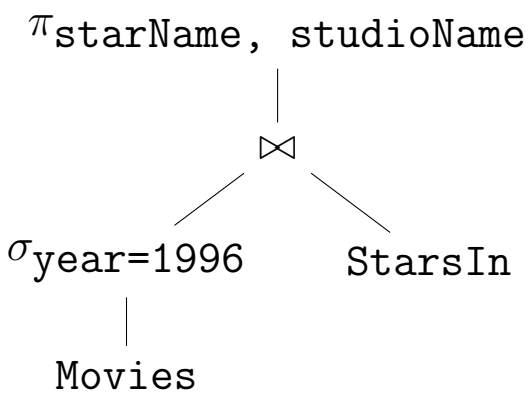
- Do selections as soon as possible
- Introduce joins instead of cartesian products
- The rule "do selections as soon as possible" is not as simple as it sounds
- The rule applies to the final plan: There are cases where one first moves selections *up* the tree, and then moves them down two branches

Example:

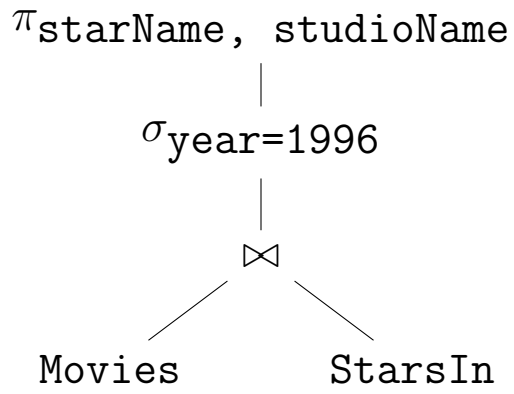
```
CREATE VIEW MoviesOf1996 AS SELECT *
   FROM Movies
   WHERE year = 1996;

SELECT starName, studioName
FROM MoviesOf1996 NATURAL JOIN StarsIn;
```

Note that `NATURAL JOIN` can be directly converted into $\bowtie$

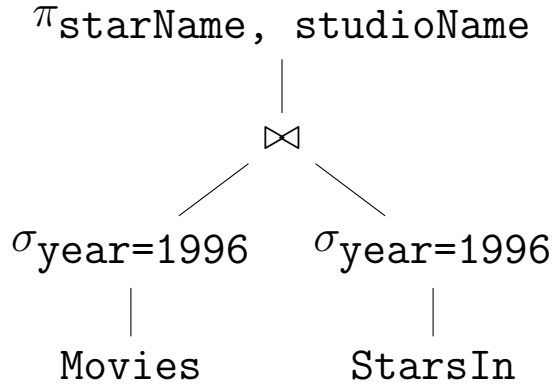
Algebraic plan for query



Move the selection first up



and then down



Which is the better plan

How do we chose between different evaluation plans?

- Certain rules, such as pushing selections down the tree, are always better
- In other cases, there is no good answer. The choice depends on the data
- We therefore need to know something about statistical properties of the data. This can be gathered by
  - Sampling of the data
  - Keeping track of previous queries
  - Domain-specific rules for the data
  - General rules that do the best possible in the absence of any of the above (this is the approach that we shall take)

Joins

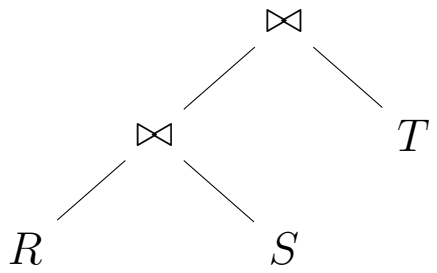
- Join is commutative and associative
- The order in which we combine relations doesn't matter
  - $R \bowtie S = S \bowtie R$
  - $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
- But the order can make a huge difference in practice

Example

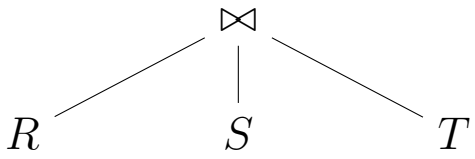
- $R(A, B)$, $S(B, C)$ and $T(C, D)$
- Suppose that each has 1.000.000 tuples
- Suppose, furthermore, that attribute B is always equal to 1 (this is an extreme case; in practice it is unlikely to be this bad) and that only one value of C in S is equal to only one value of C in T
- If we evaluate $R \bowtie S$ and then join the result to T we create a relation with 10^{12} tuples, and then join some of them to T getting 10^6 tuples
- If, on the other hand, we start with $S \bowtie T$, we get exactly one tuple, and joining it to R is much less expensive

This is an extreme example, but selection of the right join order is still very important in practice

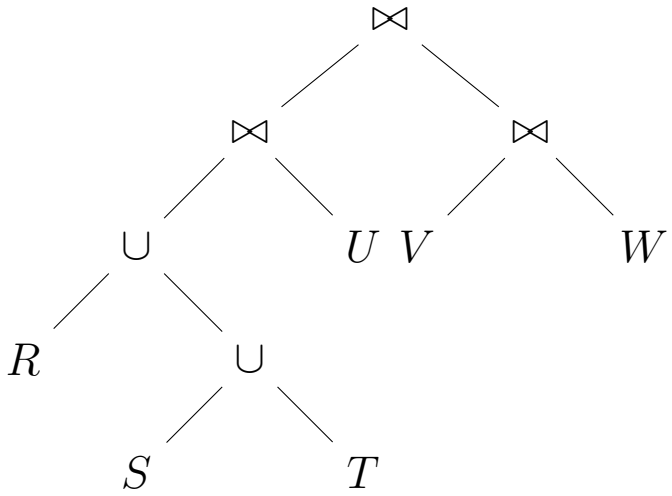
- For now, we postpone this issue
- We represent the join as an operator $\bowtie(R, S, T)$, the join of three (or more) relations
- In theory, we should also give the join conditions, but we only consider natural join
- The tree



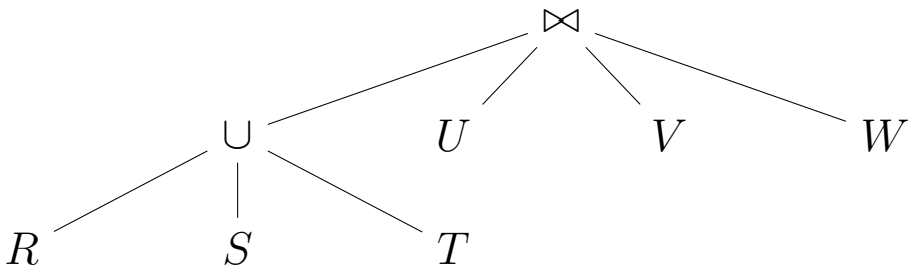
becomes



The same is done with other associative and commutative operators such as union and intersection



becomes



Search

- We have to pick the "best" tree, as discussed next
- First, a few words on *which* trees to examine
- For complex queries, examining all trees is unpractical
- There are standard heuristical techniques (such as in AI) for chosing which trees to search
- The techniques for SQL are based on these, and we only discuss search techniques for join order