Query Processing for Non-traditional Applications

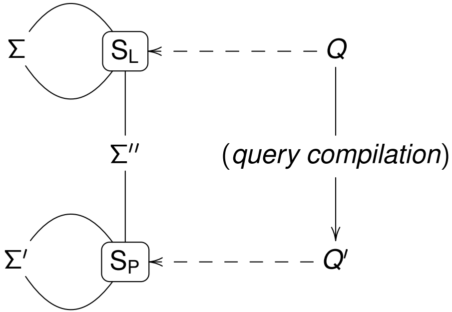
CS848 Spring 2013

Cheriton School of CS

Query Plans

Physical Design and Query Compilation: Overview

$$\Sigma = (\Sigma \cup \Sigma' \cup \Sigma'')$$



Example Physical Design: Column Store

Logical Design: $S_L = \{\texttt{employee}/3\}$

Example Physical Design: Column Store

Logical Design: $S_L = \{\texttt{employee}/3\}$

$$\forall x, y_1, y_2, z_1, z_2.\texttt{employee}(x, y_1, z_1) \wedge \texttt{employee}(x, y_2, z_2) \rightarrow (y_1 = y_2 \wedge z_1 = z_2)$$

Example Physical Design: Column Store

Logical Design: $S_L = \{\texttt{employee}/3\}$

$$\forall x, y_1, y_2, z_1, z_2.\texttt{employee}(x, y_1, z_1) \land \texttt{employee}(x, y_2, z_2) \rightarrow (y_1 = y_2 \land z_1 = z_2)$$

Physical Design: $S_P = S_A = \{\texttt{emp-rid-eid}/2/1, \texttt{emp-eid-rid}/2/0(/1)$
$\texttt{emp-rid-name}/2/1, \texttt{emp-name-rid}/2/1$
$\texttt{emp-rid-slry}/2/1, \texttt{emp-slry-rid}/2/1 \}$

Example Physical Design: Column Store

Logical Design: $S_L = \{\texttt{employee}/3\}$

$$\forall x, y_1, y_2, z_1, z_2.\texttt{employee}(x, y_1, z_1) \wedge \texttt{employee}(x, y_2, z_2) \\ \rightarrow (y_1 = y_2 \wedge z_1 = z_2)$$

Physical Design: $S_P = S_A = \{\texttt{emp-rid-eid}/2/1, \texttt{emp-eid-rid}/2/0(/1)$
$\texttt{emp-rid-name}/2/1, \texttt{emp-name-rid}/2/1$
$\texttt{emp-rid-slry}/2/1, \texttt{emp-slry-rid}/2/1 \}$

$$\forall x, y, z.\texttt{employee}(x, y, z) \rightarrow \\ \exists a.\texttt{emp-rid-eid}(a, x) \wedge \texttt{emp-rid-name}(a, y) \wedge \texttt{emp-rid-slry}(a, z)$$

Example Physical Design: Column Store

Logical Design: $S_L = \{\texttt{employee}/3\}$

$$\forall x, y_1, y_2, z_1, z_2.\texttt{employee}(x, y_1, z_1) \wedge \texttt{employee}(x, y_2, z_2) \rightarrow (y_1 = y_2 \wedge z_1 = z_2)$$

Physical Design: $S_P = S_A = \{\texttt{emp-rid-eid}/2/1, \texttt{emp-eid-rid}/2/0(/1)$
$\texttt{emp-rid-name}/2/1, \texttt{emp-name-rid}/2/1$
$\texttt{emp-rid-slry}/2/1, \texttt{emp-slry-rid}/2/1 \}$

$$\forall x, y, z.\texttt{employee}(x, y, z) \rightarrow \exists a.\texttt{emp-rid-eid}(a, x) \wedge \texttt{emp-rid-name}(a, y) \wedge \texttt{emp-rid-slry}(a, z)$$

$$\forall a, x_1, x_2.\texttt{emp-rid-eid}(a, x_1) \wedge \texttt{emp-rid-eid}(a, x_2) \rightarrow x_1 = x_2$$

$$\forall a, x.\texttt{emp-rid-eid}(a, x) \rightarrow \exists y, z.\texttt{employee}(x, y, z)$$

$$\forall a, y_1, y_2.\texttt{emp-rid-name}(a, y_1) \wedge \texttt{emp-rid-name}(a, y_2) \rightarrow y_1 = y_2$$

$$\forall a, y.\texttt{emp-rid-name}(a, y) \rightarrow \exists x.\texttt{emp-rid-eid}(a, x)$$ (same for `-slry`)

Example Physical Design: Column Store

Logical Design: $S_L = \{\texttt{employee}/3\}$

$$\forall x, y_1, y_2, z_1, z_2.\texttt{employee}(x, y_1, z_1) \wedge \texttt{employee}(x, y_2, z_2) \rightarrow (y_1 = y_2 \wedge z_1 = z_2)$$

Physical Design: $S_P = S_A = \{\texttt{emp-rid-eid}/2/1, \texttt{emp-eid-rid}/2/0(/1)$
$\texttt{emp-rid-name}/2/1, \texttt{emp-name-rid}/2/1$
$\texttt{emp-rid-slry}/2/1, \texttt{emp-slry-rid}/2/1 \}$

$$\forall x, y, z.\texttt{employee}(x, y, z) \rightarrow \exists a.\texttt{emp-rid-eid}(a, x) \wedge \texttt{emp-rid-name}(a, y) \wedge \texttt{emp-rid-slry}(a, z)$$

$$\forall a, x_1, x_2.\texttt{emp-rid-eid}(a, x_1) \wedge \texttt{emp-rid-eid}(a, x_2) \rightarrow x_1 = x_2$$

$$\forall a, x.\texttt{emp-rid-eid}(a, x) \rightarrow \exists y, z.\texttt{employee}(x, y, z)$$

$$\forall a, y_1, y_2.\texttt{emp-rid-name}(a, y_1) \wedge \texttt{emp-rid-name}(a, y_2) \rightarrow y_1 = y_2$$

$\forall a, y.\texttt{emp-rid-name}(a, y) \rightarrow \exists x.\texttt{emp-rid-eid}(a, x)$ (same for $\texttt{-slry}$)

$\forall a, x.\texttt{emp-rid-eid}(a, x) \leftrightarrow \texttt{emp-eid-rid}(x, a)$ (same for $\texttt{-name}$, $\texttt{-slry}$)

Queries over Column Store Physical Design

1. $\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y) \land \texttt{emp-rid-slry}(a, z)$

Queries over Column Store Physical Design

1. $\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y) \land \texttt{emp-rid-slry}(a, z)$

2. $\exists y, z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a)$

Queries over Column Store Physical Design

1. $\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y) \land \texttt{emp-rid-slry}(a, z)$

2. $\exists y, z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a)$

3. $\exists z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y)$

Queries over Column Store Physical Design

1. $\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y) \land \texttt{emp-rid-slry}(a, z)$
2. $\exists y, z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a)$
3. $\exists z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y)$
4. $\exists x.\texttt{employee}(x, y, z)\{y\}$

   $\{\exists a.\texttt{emp-name-rid}(y, a) \land \texttt{emp-rid-slry}(a, z)\}$

Queries over Column Store Physical Design

1. $\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y) \land \texttt{emp-rid-slry}(a, z)$
2. $\exists y, z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a)$
3. $\exists z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \land \texttt{emp-rid-name}(a, y)$
4. $\exists x.\texttt{employee}(x, y, z)\{y\}$

   $\{\exists a.\texttt{emp-name-rid}(y, a) \land \texttt{emp-rid-slry}(a, z)\}$

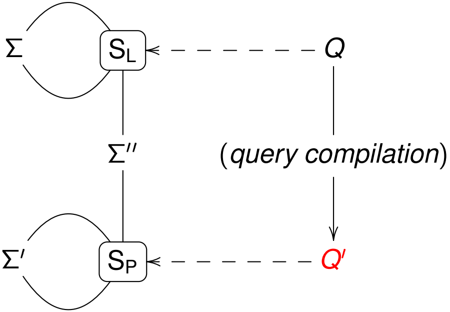
Issues to resolve (today)

- Why are the above *plans* implement the *user queries*?
- What "formulas" do qualify as *plans*?

  ⇒ how do we interpret *logical connectives* as programs?
- Are all (desired) *plans* captured by appropriate formulas?

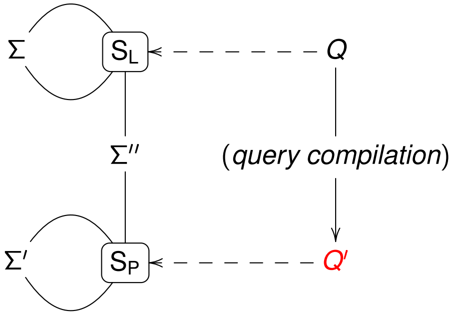
Physical Design and Query Compilation: Plans

$$\Sigma = (\Sigma \cup \Sigma' \cup \Sigma'')$$



Physical Design and Query Compilation: Plans

$$\Sigma = (\Sigma \cup \Sigma' \cup \Sigma'')$$



We consider the structure of *Q'*:

- what formulas can be interpreted as *plans*
  - ⇒ how do we deal with *SETS* in programs?
- what additional *non-logical* operations can be used.

Outline

1. Iterator Protocols to communicate Sets (review)
2. Atomic Plan Operations: Access Paths (review)
3. Logical Connectives/Quantifiers as Plan Operators
4. Beyond Logical Operators: Dealing with Duplicates

ACME Case: Access Path Code Templates

Pseudo-code templates realizing a `first/next` protocol for `emp-array0` might be given as follows (variables would be renamed for each occurrence of `emp-array0` in a query plan).

```
function emp-array0-first
  i := 0
  return emp-array0-next
```

```
function emp-array0-next
  i := i + 1
  if (i > n) return false
  x1 := emp-array[i].emp-salary
  x2 := emp-array[i].emp-num
  x3 := emp-array[i].emp-name
  return true
```

ACME Case: Access Path Code Templates

Pseudo-code templates realizing a `first/next` protocol for `emp-array0` might be given as follows (variables would be renamed for each occurrence of `emp-array0` in a query plan).

```
function emp-array0-first
  i := 0
  return emp-array0-next
```

```
function emp-array0-next
  i := i + 1
  if (i > n) return false
  x1 := emp-array[i].emp-salary
  x2 := emp-array[i].emp-num
  x3 := emp-array[i].emp-name
  return true
```

Assumes a global state recording bindings of (possible copies of) variables.

1. x_1, x_2 and x_3 to communicate the contents of `emp-array`.
2. i and n to record scanning status and size of `emp-array`.

Note: Code templates for access paths must be provided by ACME's DBA department.

Iterator Execution

Hereon, assume **C** denotes the following code that *prints a line* for each result computed by a query plan Q, where $(\text{In}(Q) \cup \text{Out}(Q)) = \{x_1, \ldots, x_m\}$.

```
if Q-first
  repeat
    println("x1" = x1, ..., "xm" = xm)
  until not Q-next
```

Iterator Execution

Hereon, assume **C** denotes the following code that *prints a line* for each result computed by a query plan Q, where $(\text{In}(Q) \cup \text{Out}(Q)) = \{x_1, \ldots, x_m\}$.

```
if Q-first
  repeat
    println("x1" = x1, ..., "xm" = xm)
  until not Q-next
```

Let Q be a query plan that scans `emp-array`.

$$\texttt{emp-array0}(x_3, x_1, x_2)$$

Iterator Execution

Hereon, assume **C** denotes the following code that *prints a line* for each result computed by a query plan Q, where $(\mathrm{In}(Q) \cup \mathrm{Out}(Q)) = \{x_1, \ldots, x_m\}$.

```
if Q-first
  repeat
    println("x1" = x1, ..., "xm" = xm)
  until not Q-next
```

Let Q be a query plan that scans `emp-array`.

$$\texttt{emp-array0}(x_3, x_1, x_2)$$

Running **C** for a database (given by interpretation) $\mathcal{I}$ produces the following output.

$$\begin{array}{l} x_1 = e_{1,1},\ x_2 = e_{1,2},\ x_3 = e_{1,3} \\ x_1 = e_{2,1},\ x_2 = e_{2,2},\ x_3 = e_{2,3} \\ \quad\vdots \\ x_1 = e_{n,1},\ x_2 = e_{n,2},\ x_3 = e_{n,3} \end{array}$$

Access Paths

The *access paths* $S_A \subseteq S_P$ are predicate symbols *associated* with a *physical capability* realized by an iterator implementation.

⇒ some attributes can be designated as *parameters*
(by convention the left-most ones)

Access Paths

The *access paths* $S_A \subseteq S_P$ are predicate symbols *associated* with a *physical capability* realized by an iterator implementation.

⇒ some attributes can be designated as *parameters*
(by convention the left-most ones)

Requirements (for access path $\mathtt{AP}/k + l/l$:

- given a *fixed values for parameters* there are only finitely many answers (bindings) to the remaining variables, i.e., the set

  $$\{a_1, \ldots, a_k \mid \mathcal{I}, \mathcal{V} \models \exists x_1, \ldots, x_l.\mathtt{AP}(x_1, \ldots, x_l, y_1, \ldots, y_k) \wedge (\bigwedge x_i = p_i)\}$$

  (where $\mathcal{V} = [y_1 = a_1, \ldots, y_k = a_k]$) is finite.
- the invocation of the iterator protocol *outputs* all and only the valuations $\mathcal{V}$ that satisfy the condition above.

Access Paths

The *access paths* $S_A \subseteq S_P$ are predicate symbols *associated* with a *physical capability* realized by an iterator implementation.

$\Rightarrow$ some attributes can be designated as *parameters*
(by convention the left-most ones)

Requirements (for access path $\mathtt{AP}/k + l/l$:

- given a *fixed values for parameters* there are only finitely many answers (bindings) to the remaining variables, i.e., the set

  $$\{a_1, \ldots, a_k \mid \mathcal{I}, \mathcal{V} \models \exists x_1, \ldots, x_l.\mathtt{AP}(x_1, \ldots, x_l, y_1, \ldots, y_k) \wedge (\bigwedge x_i = p_i)\}$$

  (where $\mathcal{V} = [y_1 = a_1, \ldots, y_k = a_k]$) is finite.
- the invocation of the iterator protocol *outputs* all and only the valuations $\mathcal{V}$ that satisfy the condition above.

A *plan interpretation* $\mathcal{I}$ satisfies the above for every access path in S_A.

(More Esoteric) Access Paths

1. Built-in "operations":
   - arithmetic (`plus`/3/2, `times`/3/2, etc.)
   - string manipulation (`concat`/3/2, `substr`/4/3, etc.)
   - ...
2. data type tests (`is-integer`/1/1)
3. pointer dereference and field extraction from records
4. (page) reads from external storage
5. ...

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$Q \;::=\; (Q_1 \wedge Q_2) \qquad (\textit{nested loop join})$$

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$
\begin{array}{rcll}
Q & ::= & (Q_1 \wedge Q_2) & (\textit{nested loop join}) \\
  & \mid & \exists x.Q, \text{ where } x \in \mathsf{V} & (\textit{duplicate preserving projection})
\end{array}
$$

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$
\begin{array}{rcll}
Q & ::= & (Q_1 \wedge Q_2) & (\textit{nested loop join}) \\
 & | & \exists x.Q, \text{ where } x \in \mathsf{V} & (\textit{duplicate preserving projection}) \\
 & | & \{Q\} & (\textit{duplicate elimination})
\end{array}
$$

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$
\begin{array}{rcll}
Q & ::= & (Q_1 \wedge Q_2) & (\textit{nested loop join}) \\
& | & \exists x.Q,\ \text{where } x \in \mathsf{V} & (\textit{duplicate preserving projection}) \\
& | & \{Q\} & (\textit{duplicate elimination}) \\
& | & [Q]_i & (\textit{cut introduction})
\end{array}
$$

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$
\begin{array}{rcll}
Q & ::= & (Q_1 \wedge Q_2) & (\textit{nested loop join}) \\
& \mid & \exists x.Q, \text{ where } x \in \mathsf{V} & (\textit{duplicate preserving projection}) \\
& \mid & \{Q\} & (\textit{duplicate elimination}) \\
& \mid & [Q]_i & (\textit{cut introduction}) \\
& \mid & !_i & (\textit{named cut})
\end{array}
$$

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$
\begin{array}{lcll}
Q & ::= & (Q_1 \wedge Q_2) & (\textit{nested loop join}) \\
 & \mid & \exists x.Q, \text{ where } x \in \mathrm{V} & (\textit{duplicate preserving projection}) \\
 & \mid & \{Q\} & (\textit{duplicate elimination}) \\
 & \mid & [Q]_i & (\textit{cut introduction}) \\
 & \mid & !_i & (\textit{named cut})
\end{array}
$$

Also require any query plan to satisfy two conditions.

1. If $Q =$ "$(Q_1 \wedge Q_2)$" then $(\mathrm{Fv}(Q_1) \cap \mathrm{Out}(Q_2)) = \emptyset$.

Conjunctive Query Plans: Syntax

The *conjunctive query plans* induced by S are as follows:

$$
\begin{array}{lcll}
Q & ::= & (Q_1 \wedge Q_2) & (\textit{nested loop join}) \\
 & \mid & \exists x.Q, \text{ where } x \in \mathsf{V} & (\textit{duplicate preserving projection}) \\
 & \mid & \{Q\} & (\textit{duplicate elimination}) \\
 & \mid & [Q]_i & (\textit{cut introduction}) \\
 & \mid & !_i & (\textit{named cut})
\end{array}
$$

Also require any query plan to satisfy two conditions.

1. If $Q =$ "$(Q_1 \wedge Q_2)$" then $(\mathrm{Fv}(Q_1) \cap \mathrm{Out}(Q_2)) = \emptyset$.
2. If $Q =$ "$\exists x.Q_1$" then $x \notin \mathrm{In}(Q_1)$.

Parameters and User Query Embeddings

Input variables, *output variables* and *user query mapping* are as follows, where $\text{Param}(\text{Uq}(Q)) = \text{In}(Q)$ always holds.

$$\text{In}(Q) = \begin{cases} \text{In}(Q_1) \cup (\text{In}(Q_2) - \text{Out}(Q_1)) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \text{In}(Q_1) & \text{if } Q = \text{“}\exists x.Q_1\text{”, “}\{Q_1\}\text{”, or “}[Q_1]_i\text{”, and} \\ \emptyset & \text{if } Q = \text{“ }!_i\text{ ”.} \end{cases}$$

$$\text{Out}(Q) = \begin{cases} \text{Out}(Q_1) \cup \text{Out}(Q_2) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \text{Out}(Q_1) \setminus \{x\} & \text{if } Q = \text{“}\exists x.Q_1\text{”}, \\ \text{Out}(Q_1) & \text{if } Q = \text{“}\{Q_1\}\text{” or “}[Q_1]_i\text{”, and} \\ \emptyset & \text{if } Q = \text{“ }!_i\text{ ”.} \end{cases}$$

$$\text{Uq}(Q) = \begin{cases} (\text{Uq}(Q_1) \wedge \text{Uq}(Q_2)) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \exists x.\,\text{Uq}(Q_1) & \text{if } Q = \text{“}\exists x.Q_1\text{”}, \\ \text{Uq}(Q_1) & \text{if } Q = \text{“}\{Q_1\}\text{” or “}[Q_1]_i\text{”, and} \\ \text{true} & \text{if } Q = \text{“ }!_i\text{ ”.} \end{cases}$$

Conjunctive Query Plans: Semantics

function $(Q_1 \wedge Q_2)$-first
  if not Q_1-first return false
  while not Q_2-first do
    if not Q_1-next return false
  return true

function $(Q_1 \wedge Q_2)$-next
  if Q_2-next return true
  while Q_1-next do
    if Q_2-first return true
  return false

---

function $(\exists x.Q_1)$-first
  return Q_1-first

function $(\exists x.Q_1)$-next
  return Q_1-next

---

function $\{Q_1\}$-first
  if not exists store S
    create S
  if Q_1-first
    empty S
    add $\langle x_1, \ldots, x_n \rangle$ *to S*
    return true
  return false

function $\{Q_1\}$-next
  while Q_1-next do
    if not $\langle x_1, \ldots, x_n \rangle \in S$
      add $\langle x_1, \ldots, x_n \rangle$ *to S*
      return true
  return false

Conjunctive Query Plans: Semantics

```
function ([Q1]i)-first        function ([Q1]i)-next
  cuti := false                 if cuti return false
  return Q1-first               return Q1-next
```

```
function (!i)-first           function (!i)-next
  cuti := true                  return false
  return true
```

Comparing and Assigning

Hereon, we assume a given physical signature includes two additional access paths.

$$\{\texttt{compare}/2/2, \texttt{assign}/2/1\} \subseteq \mathsf{S_A}$$

Comparing and Assigning

Hereon, we assume a given physical signature includes two additional access paths.

$$\{\texttt{compare}/2/2, \texttt{assign}/2/1\} \subseteq S_A$$

Semantics of compare(x_1, x_2) and assign(x_1, x_2) is given as follows.

```
function compare-first        function compare-next
  if x1 = x2 return true        return false
  return false
```

```
function assign-first         function assign-next
  x2 := x1                      return false
  return true
```

Comparing and Assigning

Hereon, we assume a given physical signature includes two additional access paths.

$$\{\texttt{compare}/2/2, \texttt{assign}/2/1\} \subseteq \mathsf{S}_\mathsf{A}$$

Semantics of $\texttt{compare}(x_1, x_2)$ and $\texttt{assign}(x_1, x_2)$ is given as follows.

```
function compare-first          function compare-next
  if x1 = x2 return true          return false
  return false
```

```
function assign-first           function assign-next
  x2 := x1                        return false
  return true
```

Also assume any theory Σ includes the following.

$$\forall x_1, x_2.\texttt{compare}(x_1, x_2) \equiv \texttt{assign}(x_1, x_2) \equiv (x_1 \approx x_2)$$

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` *x and* `name` *y of each* `employee` *that has a given* `salary` *z.*

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` *x and* `name` *y of each* `employee` *that has a given* `salary` *z.* A query plan using access path `emp-array0` can now be formulated.

$$\exists u.(\texttt{emp-array0}(u, x, y) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{z\}$ and $\{x, y\}$, respectively.

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` x *and* `name` y *of each* `employee` *that has a given* `salary` z*.* A query plan using access path `emp-array0` can now be formulated.

$$\exists u.(\texttt{emp-array0}(u, x, y) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{z\}$ and $\{x, y\}$, respectively.

Execution proceeds as follows.

1. Use access path `emp-array0` to scan `emp-array` (an atomic query subplan).

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` *x and* `name` *y of each* `employee` *that has a given* `salary` *z.* A query plan using access path `emp-array0` can now be formulated.

$$\exists u.(\texttt{emp-array0}(u, x, y) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{z\}$ and $\{x, y\}$, respectively.

Execution proceeds as follows.

1. Use access path `emp-array0` to scan `emp-array` (an atomic query subplan).
2. For each element of `emp-array` returned by this scan, compare the `salary` field with the supplied parameter value z (by using an operator for nested cross product coupled with another atomic query subplan using access path `compare`).

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` *x and* `name` *y of each* `employee` *that has a given* `salary` *z.* A query plan using access path `emp-array0` can now be formulated.

$$\exists u.(\texttt{emp-array0}(u, x, y) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{z\}$ and $\{x, y\}$, respectively.

Execution proceeds as follows.

1. Use access path `emp-array0` to scan `emp-array` (an atomic query subplan).
2. For each element of `emp-array` returned by this scan, compare the `salary` field with the supplied parameter value z (by using an operator for nested cross product coupled with another atomic query subplan using access path `compare`).
3. If this comparison evaluates to `true` then add the contents of this element to the result.

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` *x and* `name` *y of each* `employee` *that has a given* `salary` *z.* A query plan using access path `emp-array0` can now be formulated.

$$\exists u.(\texttt{emp-array0}(u, x, y) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{z\}$ and $\{x, y\}$, respectively.

Execution proceeds as follows.

1. Use access path `emp-array0` to scan `emp-array` (an atomic query subplan).
2. For each element of `emp-array` returned by this scan, compare the `salary` field with the supplied parameter value z (by using an operator for nested cross product coupled with another atomic query subplan using access path `compare`).
3. If this comparison evaluates to `true` then add the contents of this element to the result.

Note: (a) Duplicate preserving projection has no effect on execution

ACME Case: Scanning and Selection

Consider where the APS department needs a plan that will *report the* `employee-number` *x and* `name` *y of each* `employee` *that has a given* `salary` *z.* A query plan using access path `emp-array0` can now be formulated.

$$\exists u.(\texttt{emp-array0}(u, x, y) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{z\}$ and $\{x, y\}$, respectively.

Execution proceeds as follows.

1. Use access path `emp-array0` to scan `emp-array` (an atomic query subplan).
2. For each element of `emp-array` returned by this scan, compare the `salary` field with the supplied parameter value z (by using an operator for nested cross product coupled with another atomic query subplan using access path `compare`).
3. If this comparison evaluates to `true` then add the contents of this element to the result.

Note: (a) Duplicate preserving projection has no effect on execution, and (b) duplicate elimination is not required since employees must have unique employee numbers.

ACME Case: Scanning and Cutting

Consider where the APS department needs a plan that will *report the* `name` *y of any* `employee` *that has a given* `salary` *z and* `employee-number` *x.*

ACME Case: Scanning and Cutting

Consider where the APS department needs a plan that will *report the* `name` y *of any* `employee` *that has a given* `salary` z *and* `employee-number` x*.* A query plan Q using access path `emp-array0` can also be formulated.

$$\exists u, v.(\texttt{emp-array0}(u, v, y) \wedge \texttt{compare}(x, v) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{y\}$ and $\{x, z\}$, respectively.

ACME Case: Scanning and Cutting

Consider where the APS department needs a plan that will *report the* `name` y *of any* `employee` *that has a given* `salary` z *and* `employee-number` x*.* A query plan Q using access path `emp-array0` can also be formulated.

$$\exists u, v.(\texttt{emp-array0}(u, v, y) \wedge \texttt{compare}(x, v) \wedge \texttt{compare}(z, u))$$

Input and output variables: $\{y\}$ and $\{x, z\}$, respectively. Pseudo-code templates realizing a `first/next` protocol for Q is then defined as follows.

```
function Q-first
  i := 0
  while i < n do
    i := i + 1
    u := emp-array[i].emp-salary
    v := emp-array[i].emp-num
    y := emp-array[i].emp-name
    if x = v
      if z = u return true
  return false
```

```
function Q-next
  while i < n do
    i := i + 1
    u := emp-array[i].emp-salary
    v := emp-array[i].emp-num
    y := emp-array[i].emp-name
    if x = v
      if z = u return true
  return false
```

ACME Case: Scanning and Cutting

Recall that employees have unique employee numbers.

ACME Case: Scanning and Cutting

Recall that employees have unique employee numbers. Can therefore add cut introduction and named cut operators to Q to improve performance.

$$\exists u, v.([\texttt{emp-array0}(u, v, y) \wedge \texttt{compare}(x, v)]_1 \wedge\ !_1 \wedge \texttt{compare}(z, u))$$

ACME Case: Scanning and Cutting

Recall that employees have unique employee numbers. Can therefore add cut introduction and named cut operators to Q to improve performance.

$$\exists u, v.([\texttt{emp-array0}(u, v, y) \wedge \texttt{compare}(x, v)]_1 \wedge\ !_1 \wedge \texttt{compare}(z, u))$$

Pseudo-code templates realizing a `first/next` protocol for Q are then modified as follows.

```
function Q-first
  i := 0
  while i < n do
    i := i + 1
    u := emp-array[i].emp-salary
    v := emp-array[i].emp-num
    y := emp-array[i].emp-name
    if x = v
      if z = u return true
      return false
  return false
```

```
function Q-next
  return false
```

ACME Case: Eliminating Duplicates

Consider where the APS department needs a plan that will *find integers z that occur as the salary value for some* `employee`*.*

ACME Case: Eliminating Duplicates

Consider where the APS department needs a plan that will *find integers z that occur as the salary value for some* `employee`*.* A user query that specifies this requirement for OPTION 1 is as follows.

$$\exists x, y.\texttt{employee}(x, y, z)$$

ACME Case: Eliminating Duplicates

Consider where the APS department needs a plan that will *find integers z that occur as the salary value for some* `employee`*.* A user query that specifies this requirement for OPTION 1 is as follows.

$$\exists x, y.\texttt{employee}(x, y, z)$$

A query plan using access path `emp-array0` implementing this query is as follows.

$$\{\exists x, y \, . \, \texttt{emp-array0}(z, x, y)\}$$

ACME Case: Eliminating Duplicates

Consider where the APS department needs a plan that will *find integers z that occur as the salary value for some* `employee`*.* A user query that specifies this requirement for OPTION 1 is as follows.

$$\exists x, y.\texttt{employee}(x, y, z)$$

A query plan using access path `emp-array0` implementing this query is as follows.

$$\{\exists x, y\,.\,\texttt{emp-array0}(z, x, y)\}$$

The plan uses a top-level duplicate elimination operation that will use temporary store to accumulate new salary values as they occur in a scan of `emp-array` with access path `emp-array0`.

General Query Plans: Syntax

The *query plans* induced by S add two final productions:

$$Q \quad ::= \quad (Q_1 \vee Q_2) \quad (\textit{concatenation})$$

General Query Plans: Syntax

The *query plans* induced by S add two final productions:

$$
\begin{array}{rcll}
Q & ::= & (Q_1 \vee Q_2) & (\textit{concatenation}) \\
  & \mid & \neg Q & (\textit{simple complement})
\end{array}
$$

General Query Plans: Syntax

The *query plans* induced by S add two final productions:

$$\begin{array}{lcll} Q & ::= & (Q_1 \vee Q_2) & (\textit{concatenation}) \\ & \mid & \neg Q & (\textit{simple complement}) \end{array}$$

Also require any query plan to satisfy two additional conditions.

1. If $Q =$ “$(Q_1 \vee Q_2)$” then $\text{Out}(Q_1) = \text{Out}(Q_2)$.

General Query Plans: Syntax

The *query plans* induced by S add two final productions:

$$\begin{array}{lcll} Q & ::= & (Q_1 \vee Q_2) & (\textit{concatenation}) \\ & | & \neg Q & (\textit{simple complement}) \end{array}$$

Also require any query plan to satisfy two additional conditions.

1. If $Q =$ "$(Q_1 \vee Q_2)$" then $\text{Out}(Q_1) = \text{Out}(Q_2)$.
2. If $Q =$ "$\neg Q_1$" then $\text{Out}(Q_1) = \emptyset$.

Parameters and User Query Embeddings

Input variables, *output variables* and *user query mapping* are extended as follows, where $\text{Param}(\text{Uq}(Q)) = \text{In}(Q)$ always holds.

$$\text{In}(Q) = \begin{cases} \text{In}(Q_1) \cup \text{In}(Q_2) & \text{if } Q = \text{``}(Q_1 \vee Q_2)\text{''}, \text{ and} \\ \text{In}(Q_1) & \text{if } Q = \text{``}\neg Q_1\text{''}. \end{cases}$$

$$\text{Out}(Q) = \begin{cases} \text{Out}(Q_1) \cap \text{Out}(Q_2) & \text{if } Q = \text{``}(Q_1 \vee Q_2)\text{''}, \text{ and} \\ \emptyset & \text{if } Q = \text{``}\neg Q_1\text{''}. \end{cases}$$

$$\text{Uq}(Q) = \begin{cases} (\text{Uq}(Q_1) \vee \text{Uq}(Q_2)) & \text{if } Q = \text{``}(Q_1 \vee Q_2)\text{''}, \text{ and} \\ \neg\, \text{Uq}(Q_1) & \text{if } Q = \text{``}\neg Q_1\text{''}. \end{cases}$$

General Query Plans: Semantics

```
function (Q1 ∨ Q2)-first
  (Q1 ∨ Q2)-flag := true
  if Q1-first return true
  (Q1 ∨ Q2)-flag := false
  return Q2-first
```

```
function (Q1 ∨ Q2)-next
  if (Q1 ∨ Q2)-flag
    if Q1-next return true
  (Q1 ∨ Q2)-flag := false
  return Q2-next
```

```
function (¬Q1)-first
  if Q1-first return false
  return true
```

```
function (¬Q1)-next
  return false
```

ACME Case: Concatenation

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `salary` *matching either the parameter p_1 or the parameter p_2.*

ACME Case: Concatenation

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `salary` *matching either the parameter p_1 or the parameter p_2.*

A query plan using access path `emp-array0` implementing this query is as follows.

$$\{(\exists y, z.(\texttt{emp-array0}(z, x, y) \land \texttt{compare}(p_1, z)) \lor \exists u, v.(\texttt{emp-array0}(v, x, u) \land \texttt{compare}(p_2, v)))\}$$

ACME Case: Concatenation

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `salary` *matching either the parameter p_1 or the parameter p_2.*

A query plan using access path `emp-array0` implementing this query is as follows.

$$\{(\exists y, z.(\texttt{emp-array0}(z, x, y) \land \texttt{compare}(p_1, z)) \lor \exists u, v.(\texttt{emp-array0}(v, x, u) \land \texttt{compare}(p_2, v))) \}$$

An execution proceeds as follows.

1. Scan `emp-array` and add the employee number of any employee with a salary given by input parameter p_1 to a temporary store S if not already there.

ACME Case: Concatenation

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `salary` *matching either the parameter p_1 or the parameter p_2.*

A query plan using access path `emp-array0` implementing this query is as follows.

$$\{(\exists y, z.(\texttt{emp-array0}(z, x, y) \land \texttt{compare}(p_1, z)) \lor \exists u, v.(\texttt{emp-array0}(v, x, u) \land \texttt{compare}(p_2, v)))\}$$

An execution proceeds as follows.

1. Scan `emp-array` and add the employee number of any employee with a salary given by input parameter p_1 to a temporary store S if not already there.
2. Scan `emp-array` for a second time and add the employee number of any employee with a salary given by input parameter p_2 to S if not already there.

ACME Case: Concatenation

Note: The user query mapping of the plan is a *positive query* (since it is a union of two conjunctive queries).

ACME Case: Concatenation

Note: The user query mapping of the plan is a *positive query* (since it is a union of two conjunctive queries).

An alternative plan can be formulated that avoids the need for two scans of `emp-array`.

$$\{\exists y, z.(\texttt{emp-array0}(z, x, y) \wedge (\texttt{compare}(p_1, z) \vee \texttt{compare}(p_2, z)))\}$$

The plan illustrates a common plan idiom for determining if a given value occurs in a given small fixed set of values.

ACME Case: Concatenation

Note: The user query mapping of the plan is a *positive query* (since it is a union of two conjunctive queries).

An alternative plan can be formulated that avoids the need for two scans of `emp-array`.

$$\{\exists y, z.(\texttt{emp-array0}(z, x, y) \wedge (\texttt{compare}(p_1, z) \vee \texttt{compare}(p_2, z)))\}$$

The plan illustrates a common plan idiom for determining if a given value occurs in a given small fixed set of values.

Two possible reasons for requiring top-level duplicate elimination.

1. Individual employee numbers may be related to more than one salary or employee name.

ACME Case: Concatenation

Note: The user query mapping of the plan is a *positive query* (since it is a union of two conjunctive queries).

An alternative plan can be formulated that avoids the need for two scans of `emp-array`.

$$\{\exists y, z.(\texttt{emp-array0}(z, x, y) \wedge (\texttt{compare}(p_1, z) \vee \texttt{compare}(p_2, z)))\}$$

The plan illustrates a common plan idiom for determining if a given value occurs in a given small fixed set of values.

Two possible reasons for requiring top-level duplicate elimination.

1. Individual employee numbers may be related to more than one salary or employee name.
2. Parameters p_1 and p_2 may not be distinct.

ACME Case: Concatenation

The first reason is ruled out by the logical design of `payroll`.

ACME Case: Concatenation

The first reason is ruled out by the logical design of `payroll`.

The second reason can be ruled out by modifying the plan to ensure the second subplan for the concatenation operator will only return additional results when p_1 and p_2 are distinct.

$$\begin{aligned}&\exists y, z.(\texttt{emp-array0}(z, x, y)\\ &\quad \wedge (\texttt{compare}(p_1, z) \vee (\neg\, \texttt{compare}(p_1, p_2) \wedge \texttt{compare}(p_2, z))))\end{aligned}$$

ACME Case: Concatenation

The first reason is ruled out by the logical design of `payroll`.

The second reason can be ruled out by modifying the plan to ensure the second subplan for the concatenation operator will only return additional results when p_1 and p_2 are distinct.

$$\exists y, z.(\texttt{emp-array0}(z, x, y) \\ \quad \wedge (\texttt{compare}(p_1, z) \vee (\neg\, \texttt{compare}(p_1, p_2) \wedge \texttt{compare}(p_2, z))))$$

Note that the query mapping for this plan is no longer a positive query.

ACME Case: Concatenation

The first reason is ruled out by the logical design of `payroll`.

The second reason can be ruled out by modifying the plan to ensure the second subplan for the concatenation operator will only return additional results when p_1 and p_2 are distinct.

$$\exists y, z.(\texttt{emp-array0}(z, x, y) \\ \wedge (\texttt{compare}(p_1, z) \vee (\neg\, \texttt{compare}(p_1, p_2) \wedge \texttt{compare}(p_2, z))))$$

Note that the query mapping for this plan is no longer a positive query.

An alternative plan avoids simple complement by using a cut.

$$\exists y, z.(\texttt{emp-array0}(z, x, y) \wedge [(\texttt{compare}(p_1, z) \vee \texttt{compare}(p_2, z))]_1 \wedge\, !_1))$$

ACME Case: Simple Complement

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `name` *that is also unique.*

ACME Case: Simple Complement

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `name` *that is also unique.*

Two plans using access path `emp-array0` and a simple complement operator are as follows.

$$
\begin{aligned}
&\exists y, z.(\texttt{emp-array0}(z, x, y) \\
&\quad \land \neg \exists u, v, w.(\texttt{emp-array0}(u, v, w) \\
&\qquad \land \texttt{compare}(y, w) \\
&\qquad \land \neg \texttt{compare}(x, v)))
\end{aligned}
$$

$$
\begin{aligned}
&\exists y, z.(\texttt{emp-array0}(z, x, y) \\
&\quad \land \neg \exists u, v, w.([\texttt{emp-array0}(u, v, w)]_1 \\
&\qquad \land \texttt{compare}(y, w) \\
&\qquad \land \neg \texttt{compare}(x, v)) \\
&\qquad \land\ !_1)
\end{aligned}
$$

ACME Case: Simple Complement

Consider where the APS department needs a plan that will *find the* `employee-number` *x for any* `employee` *that has a* `name` *that is also unique.*

Two plans using access path `emp-array0` and a simple complement operator are as follows.

$$\begin{aligned}
&\exists y, z.(\texttt{emp-array0}(z, x, y) \\
&\quad \land \neg \exists u, v, w.(\texttt{emp-array0}(u, v, w) \\
&\qquad \land \texttt{compare}(y, w) \\
&\qquad \land \neg\, \texttt{compare}(x, v)))
\end{aligned}$$

$$\begin{aligned}
&\exists y, z.(\texttt{emp-array0}(z, x, y) \\
&\quad \land \neg \exists u, v, w.([\texttt{emp-array0}(u, v, w)]_1 \\
&\qquad \land \texttt{compare}(y, w) \\
&\qquad \land \neg\, \texttt{compare}(x, v)) \\
&\qquad \land\ !_1)
\end{aligned}$$

Question: Which is more efficient?

When do Plans Implement User Queries?

Requirements

Given a *plan interpretation* $\mathcal{I}$, the plan Q' (driven by **C**) outputs exactly the valuations that make the *user query* Q true in $\mathcal{I}$.

When do Plans Implement User Queries?

Requirements

Given a *plan interpretation* $\mathcal{I}$, the plan Q' (driven by **C**) outputs exactly the valuations that make the *user query* Q true in $\mathcal{I}$.

What can we do with this?

- we show how to *model* many (if not most) features of standard SQL implementations using the operators introduced today with the help of creative physical design and selection of access paths (next time).

When do Plans Implement User Queries?

Requirements

Given a *plan interpretation* $\mathcal{I}$, the plan Q' (driven by **C**) outputs exactly the valuations that make the *user query* Q true in $\mathcal{I}$.

What can we do with this?

- we show how to *model* many (if not most) features of standard SQL implementations using the operators introduced today with the help of creative physical design and selection of access paths (next time).

How do we *actually* find plans?

1. we search for Q' that (as a formula) *is logically equivalent* to Q under the logical and physical schama constraints Σ, and
2. we *improve* Q' by eliminating *duplicate elimination operations* and by inserting *cuts*.

Queries over Column Store Physical Design Revisited

1. $\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \wedge \texttt{emp-rid-name}(a, y) \wedge \texttt{emp-rid-slry}(a, z)$
2. $\exists y, z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a)$
3. $\exists z.\texttt{employee}(x, y, z)$

   $\exists a.\texttt{emp-eid-rid}(x, a) \wedge \texttt{emp-rid-name}(a, y)$
4. $\exists x.\texttt{employee}(x, y, z)\{y\}$

   $\{\exists a.\texttt{emp-name-rid}(y, a) \wedge \texttt{emp-rid-slry}(a, z)\}$

Results:

- Plans for queries 1-4 are logically equivalent to the given user queries.
- Plans 1-3 can avoid duplicate eimination operator.

Related Issues

- Relational Algebra
- Domain Independence and Range restricted Queries
- Temporary storage
- Ordered properties of iterated semantics (merge joins?)
- Streaming queries