Logical Approach to Physical Data Independence and Query Compilation

Query Compilation

David Toman

D.R. Cheriton School of Computer Science

University of Waterloo

The Story So Far...

1. Physical Data Independence (OBDA, Data Exchange, ...)
2. Logic-based formalization (Relational model, constraints)
3. Queries and Answers

$$\mathrm{cert}_{\Sigma,D}(\varphi) = \{\vec{a} \mid \Sigma \cup D \models \varphi(\vec{a})\} = \bigcap_{I \models \Sigma \cup D} \{\vec{a} \mid I \models \varphi(\vec{a})\}$$

The Story So Far…

1. Physical Data Independence (OBDA, Data Exchange, …)
2. Logic-based formalization (Relational model, constraints)
3. Queries and Answers

$$\text{cert}_{\Sigma,D}(\varphi) = \{\vec{a} \mid \Sigma \cup D \models \varphi(\vec{a})\} = \bigcap_{I \models \Sigma \cup D} \{\vec{a} \mid I \models \varphi(\vec{a})\}$$

Difficulties

1. $\vec{a} \in \text{cert}_{\Sigma,D}(\varphi)$ undecidable for FOL
   $\Rightarrow$ *Data Complexity* (in $|D|$) is high for most decidable fragments
   $\Rightarrow$ *Lite Description Logics* and *CQ*—sacrificing expressiveness
2. Unintuitive Answers
   $\Rightarrow$ "does John have a phone #?" $\rightarrow$ yes; "what is john's phone #?" $\rightarrow$ { }
   $\Rightarrow$ non-compositional (no algebra for certain answers)
3. Efficient algorithm only for range-restricted queries over *closed KB*.

Query Compilation

Equivalent Range-restricted Queries

IDEA

Restrict allowed *user queries* to those that are *logically equivalent* to a *range-restricted query over* S_A under Σ.

Equivalent Range-restricted Queries

IDEA

Restrict allowed *user queries* to those that are *logically equivalent* to a *range-restricted query over* $\mathsf{S_A}$ under Σ.

Separates execution if *user query* φ into two steps:

1. finding ψ over $\mathsf{S_A}$ such that $\Sigma \models \varphi \leftrightarrow \psi$ (compilation), and
2. executing ψ over D_{A} (execution)

Equivalent Range-restricted Queries

IDEA

Restrict allowed *user queries* to those that are *logically equivalent* to a *range-restricted query over* S_A under Σ.

Separates execution if *user query* φ into two steps:

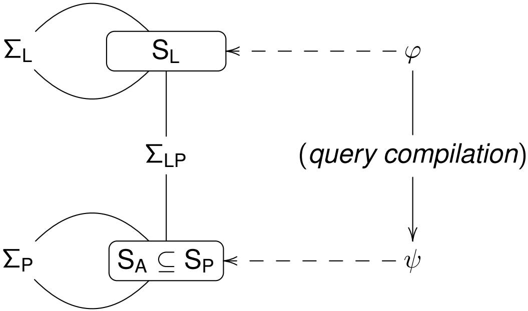
1. finding ψ over S_A such that $\Sigma \models \varphi \leftrightarrow \psi$ (compilation), and
2. executing ψ over D_A (execution)

Intuition(s)

- The user *has the illusion* there is *exactly one* explicit *closed-world* database instance over S_L;
- The system has a choice of *different* D_A instances to support this: record id-s/pointers, record distribution/fragmentation, ...

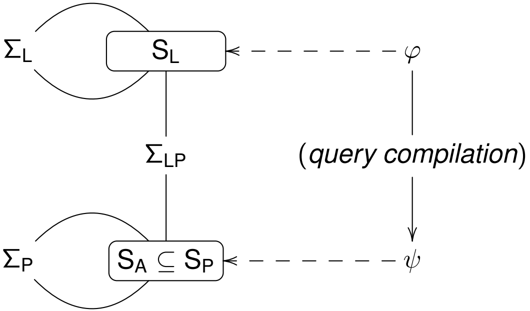
Physical Design and Query Compilation: Overview

$$\Sigma = (\Sigma_L \cup \Sigma_{LP} \cup \Sigma_P)$$



Physical Design and Query Compilation: Overview

$$\Sigma = (\Sigma_L \cup \Sigma_{LP} \cup \Sigma_P)$$



Issues

1. how to find ψ, given φ, Σ, and S_A? (and how hard is this?)
2. how closely does ψ describe actual execution? (and what is left out?)

Case Study: Relational Systems Implementation

Case Study: RDBMs Internals

Physical Design Desiderata (v0)

- arbitrary vertical fragmentation of relations (up to a *column store*)
- indexing (primary and secondary)
- ...

Case Study: RDBMs Internals

Physical Design Desiderata (v0)

- arbitrary vertical fragmentation of relations (up to a *column store*)
- indexing (primary and secondary)
- ...

IDEA: Record IDs

Every tuple in an instance of a relation is *uniquely* tagged by a *record ID*.

$\Rightarrow$ typically, given a RID, the corresponding tuple can be located “efficiently”

Case Study: RDBMs Internals

Physical Design Desiderata (v0)

- arbitrary vertical fragmentation of relations (up to a *column store*)
- indexing (primary and secondary)
- ...

IDEA: Record IDs

Every tuple in an instance of a relation is *uniquely* tagged by a *record ID*.

$\Rightarrow$ typically, given a RID, the corresponding tuple can be located "efficiently"

- vertical fragmentation: fragments contain RID (losses join);
- primary index: search key "clusterd" with RID (typically stores tuples);
- secondary index: search keys and RIDs

$\Rightarrow$ all translates to a *selection of access paths* (and constraints)

Example

Logical Schema:

- ternary user relation $A/3$.

Physical Schema:

- a 4-ary *base file* $EAs/4/0$ (A-scan)
  $\Rightarrow A(x, y, z) \leftrightarrow (\exists a.EAs(a, x, y, z))$
  $\Rightarrow a$ is a key for EAs.
- a 4-ary *base file* $EAr/4/1$ (A-ref)
  $\Rightarrow EAs(a, x, y, z) \leftrightarrow EAr(a, x, y, z)$
- an $EAIx/2/1$ index on A's attribute x
  $\Rightarrow EAIx(x, a) \leftrightarrow (\exists y, z.EAs(a, x, y, z))$
- an $EAIy/2/1$ index on A's attribute y
  $\Rightarrow EAIx(y, a) \leftrightarrow (\exists x, z.EAs(a, x, y, z))$

Example: Access Paths

```
(defap A-scan :logical AB  :name EAs
    :in   []         :out  [a :int x :int y :int z :int]
    :cost ["10n" "10n"])

(defap A-ref  :logical AB  :name EAr
    :in   [a :int]   :out  [x :int y :int z :int]
    :cost ["10000log(n)" "1"])

(defap A-idx1 :logical AI1 :name EAIx
    :in   [x :int]   :out  [a :int]
    :cost ["log(n)" "log(n)"])

(defap A-idx2 :logical AI2 :name EAIy
    :in   [y :int]   :out  [a :int]
    :cost ["log(n)" "log(n)"])
```

Example: Constraints

```
(defschema schema
  :constraints [[
; base table
   (fol (-> (A ?x ?y ?z) (exists [?a] (AB ?a ?x ?y ?z))))
   (fol (-> (AB ?a ?x ?y ?z) (A ?x ?y ?z )))
; index on "x"
   (fol (-> (AB ?a ?x ?y ?z) (AI1 ?x ?a)))
   (fol (-> (AI1 ?x ?a) (exists [?y ?z] (AB ?a ?x ?y ?z))))
; index on "y"
   (fol (-> (AB ?a ?x ?y ?z) (AI2 ?y ?a)))
   (fol (-> (AI2 ?y ?a) (exists [?x ?z] (AB ?a ?x ?y ?z))))
 ]
 (primary-key [a] (AB a x y z))
 ]
 :access-paths [A-scan A-ref A-idx1 A-idx2]
 :cost-model ->Complexity
)
```

Case Study: RDBMs Internals

Example

User queries (and expected execution patterns):

1. $A(x, y, z)$
2. $A(x, y, z)$ given value for x
3. $A(x, y, z)$ given value for x and y
4. $\exists y, z.A(x, y, z)$ given x
5. $\exists z.A(x, y, z)$ given value for x and y

Case Study: RDBMs Internals

Example

User queries (and expected execution patterns):

1. $A(x, y, z)$

   $\Rightarrow$ scan all tuples in A using *EAs*.

2. $A(x, y, z)$ given value for x

3. $A(x, y, z)$ given value for x and y

4. $\exists y, z.A(x, y, z)$ given x

5. $\exists z.A(x, y, z)$ given value for x and y

Case Study: RDBMs Internals

Example

User queries (and expected execution patterns):

1. $A(x, y, z)$

   $\Rightarrow$ scan all tuples in A using EAs.

2. $A(x, y, z)$ given value for x

   $\Rightarrow$ lookup x in $EAIx$ and use RID for lookup in EAr.

3. $A(x, y, z)$ given value for x and y

4. $\exists y, z.A(x, y, z)$ given x

5. $\exists z.A(x, y, z)$ given value for x and y

Case Study: RDBMs Internals

Example

User queries (and expected execution patterns):

1. $A(x, y, z)$

   $\Rightarrow$ scan all tuples in A using EAs.

2. $A(x, y, z)$ given value for x

   $\Rightarrow$ lookup x in $EAIx$ and use RID for lookup in EAr.

3. $A(x, y, z)$ given value for x and y

   $\Rightarrow$ lookup x in $EAIx$, use RID for lookup in EAr, and compare y values.

4. $\exists y, z.A(x, y, z)$ given x

5. $\exists z.A(x, y, z)$ given value for x and y

Case Study: RDBMs Internals

Example

User queries (and expected execution patterns):

1. $A(x, y, z)$
   $\Rightarrow$ scan all tuples in A using EAs.
2. $A(x, y, z)$ given value for x
   $\Rightarrow$ lookup x in $EAIx$ and use RID for lookup in EAr.
3. $A(x, y, z)$ given value for x and y
   $\Rightarrow$ lookup x in $EAIx$, use RID for lookup in EAr, and compare y values.
4. $\exists y, z.A(x, y, z)$ given x
   $\Rightarrow$ lookup x in $EAIx$.
5. $\exists z.A(x, y, z)$ given value for x and y

Case Study: RDBMs Internals

Example

User queries (and expected execution patterns):

1. $A(x, y, z)$
   $\Rightarrow$ scan all tuples in A using *EAs*.
2. $A(x, y, z)$ given value for x
   $\Rightarrow$ lookup x in *EAIx* and use RID for lookup in *EAr*.
3. $A(x, y, z)$ given value for x and y
   $\Rightarrow$ lookup x in *EAIx*, use RID for lookup in *EAr*, and compare y values.
4. $\exists y, z.A(x, y, z)$ given x
   $\Rightarrow$ lookup x in *EAIx*.
5. $\exists z.A(x, y, z)$ given value for x and y
   $\Rightarrow$ lookup x in *EAIx*, y in *EAIy* and compare retriever RIDs.

Example: Running the System

```
scan (A x y z) []
Plan: 0 (10n,10n)
E?x1.EAs(?x1,x,y,z)

lookup x: (A x y z) [x]
Plan: 1 (10000log(n)^2 + 2log(n),1log(n))
E?x1.(EAIx(x,?x1)^E?s0.(EAr(?x1,?s0,y,z)))

lookup x,y: (A x y z) [x,y]
Plan: 22 (10000log(n)^2 + 3log(n),1log(n))
E?x1.(EAIx(x,?x1)^E?s1.E?s0.(EAr(?x1,?s0,?s1,z)^Cmp(y,?s1)))

lookup x index only: (exists [?y ?z] (A x ?y ?z)) [x]
Plan: 0 (1log(n),1log(n))
E?x1.EAIx(x,?x1)

lookup x,y index only: (exists [?z] (A x y ?z)) [x,y]
Plan: 90 (2log(n)^2 + 1log(n),1log(n)^2)
E?x1.(EAIy(y,?x1)^E?s0.(EAIx(x,?s0)^Cmp(?x1,?s0)))
```

Execution Model, part 1

How can we say that, e.g.,

$$\exists x_1.(EAIy(y, x_1) \wedge \exists s_0.(EAIx(x, s_0) \wedge Cmp(x_1, s_0)))$$

implements $\exists z.A(x, y, z)$ given value for x and y efficiently?

Execution Model, part 1

How can we say that, e.g.,

$$\exists x_1.(EAly(y, x_1) \wedge \exists s_0.(EAlx(x, s_0) \wedge Cmp(x_1, s_0)))$$

implements $\exists z.A(x, y, z)$ given value for x and y efficiently?

- implements:
  $\Sigma \models \exists z.A(x, y, z) \leftrightarrow \exists x_1.(EAly(y, x_1) \wedge \exists s_0.(EAlx(x, s_0) \wedge Cmp(x_1, s_0)));$

Execution Model, part 1

How can we say that, e.g.,

$$\exists x_1.(EAly(y, x_1) \wedge \exists s_0.(EAlx(x, s_0) \wedge Cmp(x_1, s_0)))$$

implements $\exists z.A(x, y, z)$ given value for x and y efficiently?

- implements:
  $\Sigma \models \exists z.A(x, y, z) \leftrightarrow \exists x_1.(EAly(y, x_1) \wedge \exists s_0.(EAlx(x, s_0) \wedge Cmp(x_1, s_0)))$;
- efficiently—depends on mapping to actual operations:

access paths	$\mapsto$	access path access
conjunction	$\mapsto$	nested loops join
existential quantification	$\mapsto$	projection

  and a *cost model* (estimates cost of executing the query from summaries)

Example: Nested Loops Join

IDEA:

All operators implement an iterator (cursor) protocol:

- `get-first`: gets/searches for the first applicable tuple
- `get-next`: gets/searches for the next applicable tuple

Example: Nested Loops Join

IDEA:

All operators implement an iterator (cursor) protocol:

`get-first`: gets/searches for the first applicable tuple

`get-next`: gets/searches for the next applicable tuple

Nested Loops Join (NLJ):

Open and get first tuple:

```
function (Q1 ∧ Q2)-first
   if not Q1-first return false
   while not Q2-first do
      if not Q1-next return false
   return true
```

Get next tuple:

```
function (Q1 ∧ Q2)-next
   if Q2-next return true
   while Q1-next do
      if Q2-first return true
   return false
```

Range Restricted Queries Revisited

The idea of *range-restricted queries* is *codified* by specifying *Input and Output Variables* and their interactions:

Range Restricted Queries Revisited

The idea of *range-restricted queries* is *codified* by specifying *Input and Output Variables* and their interactions:

$$\text{In}(Q) = \begin{cases} \text{In}(Q_1) \cup (\text{In}(Q_2) - \text{Out}(Q_1)) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \text{In}(Q_1) & \text{if } Q = \text{“}\exists x.Q_1\text{”}, \\ \text{In}(Q_1) \cup \text{In}(Q_2) & \text{if } Q = \text{“}(Q_1 \vee Q_2)\text{”, and} \\ \text{In}(Q_1) & \text{if } Q = \text{“}\neg Q_1\text{”}. \end{cases}$$

$$\text{Out}(Q) = \begin{cases} \text{Out}(Q_1) \cup \text{Out}(Q_2) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \text{Out}(Q_1) \setminus \{x\} & \text{if } Q = \text{“}\exists x.Q_1\text{”}, \\ \text{Out}(Q_1) \cap \text{Out}(Q_2) & \text{if } Q = \text{“}(Q_1 \vee Q_2)\text{”, and} \\ \emptyset & \text{if } Q = \text{“}\neg Q_1\text{”}. \end{cases}$$

Range Restricted Queries Revisited

The idea of *range-restricted queries* is *codified* by specifying *Input and Output Variables* and their interactions:

$$\mathrm{In}(Q) = \begin{cases} \mathrm{In}(Q_1) \cup (\mathrm{In}(Q_2) - \mathrm{Out}(Q_1)) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \mathrm{In}(Q_1) & \text{if } Q = \text{“}\exists x.Q_1\text{”}, \\ \mathrm{In}(Q_1) \cup \mathrm{In}(Q_2) & \text{if } Q = \text{“}(Q_1 \vee Q_2)\text{”, and} \\ \mathrm{In}(Q_1) & \text{if } Q = \text{“}\neg Q_1\text{”}. \end{cases}$$

$$\mathrm{Out}(Q) = \begin{cases} \mathrm{Out}(Q_1) \cup \mathrm{Out}(Q_2) & \text{if } Q = \text{“}(Q_1 \wedge Q_2)\text{”}, \\ \mathrm{Out}(Q_1) \setminus \{x\} & \text{if } Q = \text{“}\exists x.Q_1\text{”}, \\ \mathrm{Out}(Q_1) \cap \mathrm{Out}(Q_2) & \text{if } Q = \text{“}(Q_1 \vee Q_2)\text{”, and} \\ \emptyset & \text{if } Q = \text{“}\neg Q_1\text{”}. \end{cases}$$

No *selection* operator: realized by

- supplying a value to a index lookup, or
- by an additional nested-loops join with a built-in table `Cmp`/2/2.

Standard RDBMs Summary

Standard (relational) Physical Design

`CREATE TABLE foo` DDL command causes

1. a *logical symbol* `foo` to be created;
2. a (disk-based) *file* `foo-file` of (appropriate) records to be created; and
3. a link between these two objects to be *recorded* (where?)

Standard RDBMs Summary

Standard (relational) Physical Design

`CREATE TABLE foo` DDL command causes

1. a *logical symbol* `foo` to be created;
2. a (disk-based) *file* `foo-file` of (appropriate) records to be created; and
3. a link between these two objects to be *recorded* (where?)
4. . . . numerous pages of additional options these days

- Record IDs and Indexing
- Multi-level store
- Horizontal partitioning
- Data replication
- Delegation to other database engines
- Materialized views and cached query results