PDQ: A PLATFORM FOR REFORMULATING QUERIES

Michael Benedikt, George Konstantinidis, Efthymia Tsamoura - Oxford University

Problem statement

Given a conjunctive query

$$Q :- \exists x_1 \ldots x_k \bigwedge_i R_i$$

over a schema with **access restrictions** over the schema relations and **integrity constraints Σ** and a **cost function C** find a plan that **minimises the cost to answer Q under C**.

Access restrictions over relations require values of certain attributes to be given as inputs to access relations.

The integrity constraints **Σ** are given by tuple-generating dependencies (TGDs):

$$\forall x_1,\ldots, x_k \bigwedge_i A_i \rightarrow \exists y_1,\ldots, y_l \bigwedge_j B_j$$

Example

Relations

Customer(cid)
CustomerDetails(cid,cname,cphone)
Product(pid,pname,pprice)
Item(itemid,pid)
Bought(cid,itemid)

Constraints

Customer(cid) → CustomerDetails(cid,cname,cphone)
CustomerDetails(cid,cname,cphone) → Customer(cid)

Access methods

Customer$^{\emptyset}$: ∅ → cid
Product$^{\emptyset}$: ∅ → pid, pname, pprice
Bought^{cid}: cid → itemid
Item^{pid}: itemid → pid

Query

SELECT CustomerDetails.cid, Product.pprice
FROM CustomerDetails, Product, Item, Bought
WHERE CustomerDetails.cid = HasBought.cid
AND Item.itemid = HasBought.itemid
AND Product.pid = Item.pid

Q:- ∃ pid, pname,itemid CustomerDetails(cid,cname,cphone), Bought(cid,itemid), Product(pid, pname, pprice), Item(itemid,pid)

Q*:- CustomerDetails(cid_0,$cname_0$, $cphone_0$), Bought(cid_0, $itemid_0$), Product(pid_0, $pname_0$, $pprice_0$), Item($itemid_0$, pid_0)

Hidden database: CustomerDetails(cid_0,$cname_0$, $cphone_0$), Bought(cid_0, $itemid_0$), Product(pid_0, $pname_0$, $pprice_0$), Item($itemid_0$, pid_0), Customer(cid_0)

Accessibility axioms

Customer(cid) → InferredAccessibleCustomer(cid) ∧ Accessible(cid)

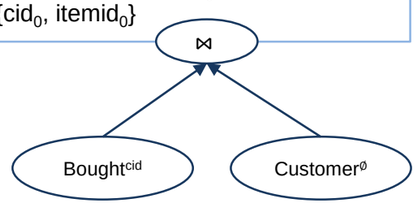
Product(pid,pname,pprice) → InferredAccessibleProduct(pid,pname,pprice) ∧ Accessible(pid) ∧ Accessible(pname) ∧ Accessible(pprice)

Accessible(pid) ∧ Item(itemid,pid) → InferredAccessibleItem(itemid,pid) ∧ Accessible(itemid)

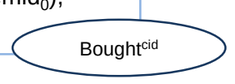
Accessible(cid) ∧ Bought(cid,itemid) → InferredAccessibleBought(cid,itemid) ∧ Accessible(itemid)

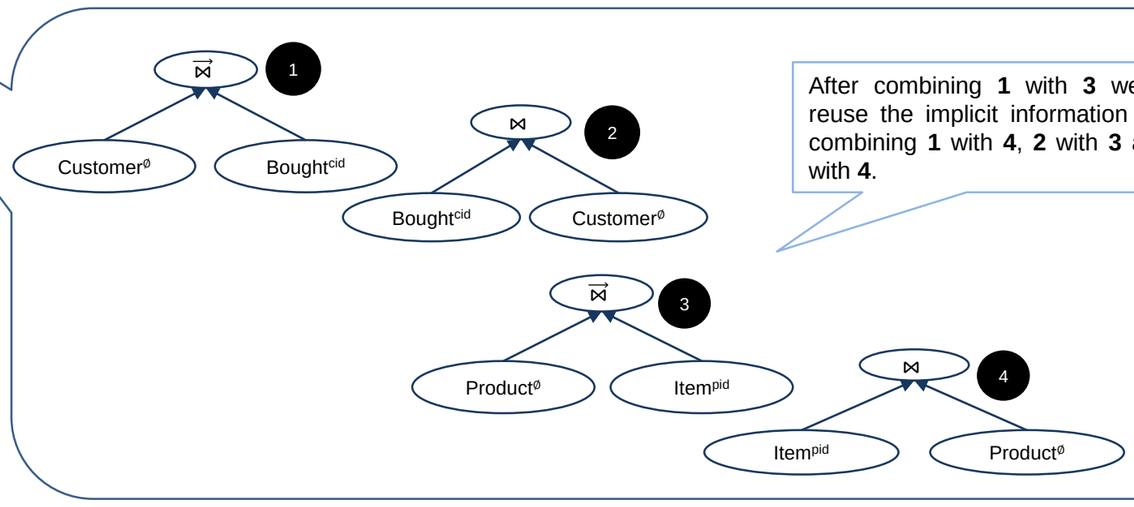
Status: Unsuccessful, closed
Inputs: ∅
Implicit information: Customer(cid_0), CustomerDetails(cid_0,$cname_0$, $cphone_0$), InferredAccessibleCustomer(cid_0), Accessible={cid_0}

Status: Unsuccessful, open
Inputs: cid↦cid_0
Implicit information: Bought(cid_0, $itemid_0$), Customer(cid_0), CustomerDetails(cid_0,$cname_0$, $cphone_0$), InferredAccessibleBought(cid_0, $itemid_0$), InferredAccessibleCustomer(cid_0), Accessible={cid_0, $itemid_0$}



Status: Unsuccessful, open
Inputs: cid↦cid_0
Implicit information: Bought(cid_0, $itemid_0$), InferredAccessibleBought(cid_0, $itemid_0$), Accessible={cid_0, $itemid_0$}





Building up plans

Access plans

Plan language
- ❑ **Access operators**
- ❑ **Dependent join operators**
- ❑ **Joins, selections and projections**

Each plan is associated with
- ❑ **the inputs required to perform the plan**
- ❑ **implicit information** found through **reasoning**

A plan can be
- ❑ **Open** (requires inputs to run) or **closed** (can be run stand alone)
- ❑ **Successful** (equivalent to query) or **unsuccessful**

Deriving implicit information

Preprocessing step
- ❑ Create "hidden database" of facts by forming canonical database of **Q Q*** and taking consequences under **Σ**.
- ❑ Augment **Σ** with accessibility axioms and inferred accessible copies of the constraints in **Σ**.

Implicit information of accesses
- ❑ consequences of hidden facts exposed by operators

Implicit information of BinaryOperator(Plan$_1$,Plan$_2$)
- ❑ consequences of [implicit information of Plan$_1$ ∪ implicit information of Plan$_2$] under augmented **Σ**

Pruning out sub-plans

Keep the best plan within some class
- ❑ group plans by input and by implicit information
- ❑ discard plans with the same or less implicit information and higher cost

Prune out unpromising sub-plans
- ❑ prune out plans with cost higher than the best closed and successful plan found

Speeding-up planning

- ❑ Search the plan space in parallel
- ❑ Speeding up reasoning
  - group plans based on their implicit information
  - when combining plans from two groups reason only the first time a composite plan is created from these two groups
- ❑ Prune out plans with cost higher than the best plan prior to reasoning

Prototype with LogicBlox

LogicBlox is a relational database geared toward analytics and predictions.

LogicBlox programs are implemented in the proprietary language LogicQL, derived from Datalog.

A PDQ server instance is started upon the creation or opening of an LB workspace.

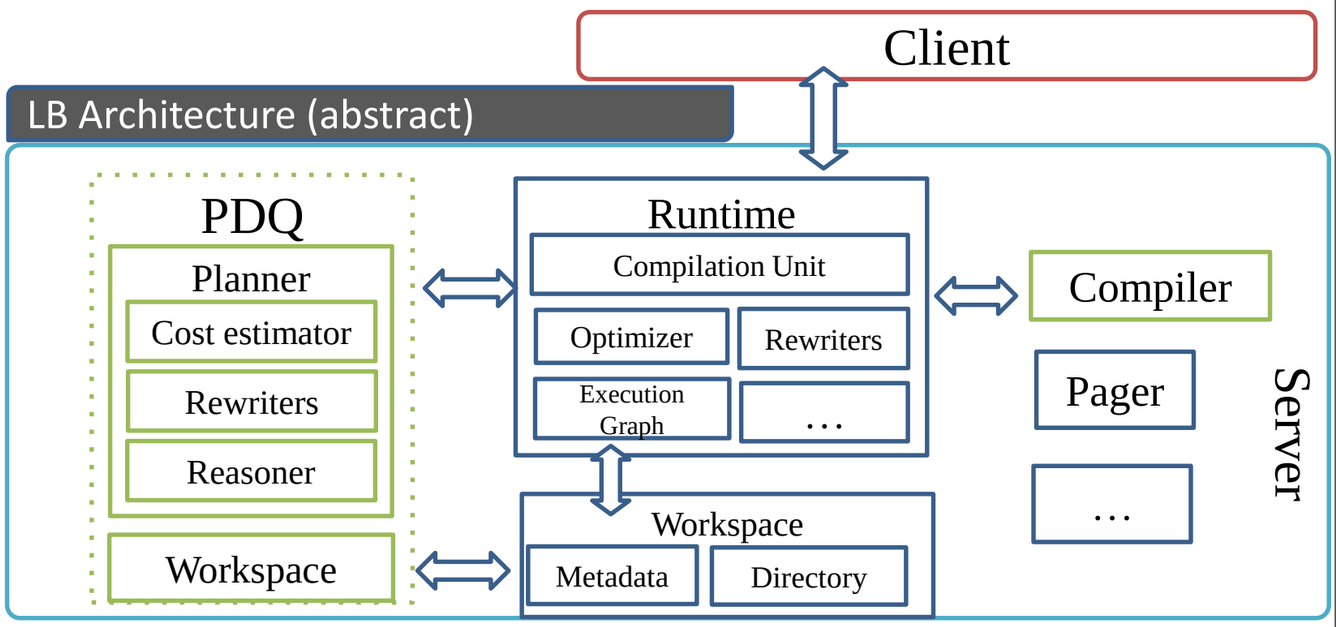
PDQ is initialized with all relevant information from the workspace such as **views** and **constraints**.

PDQ **optimizes** LB rules, by offering equivalent rewritings with different cost.

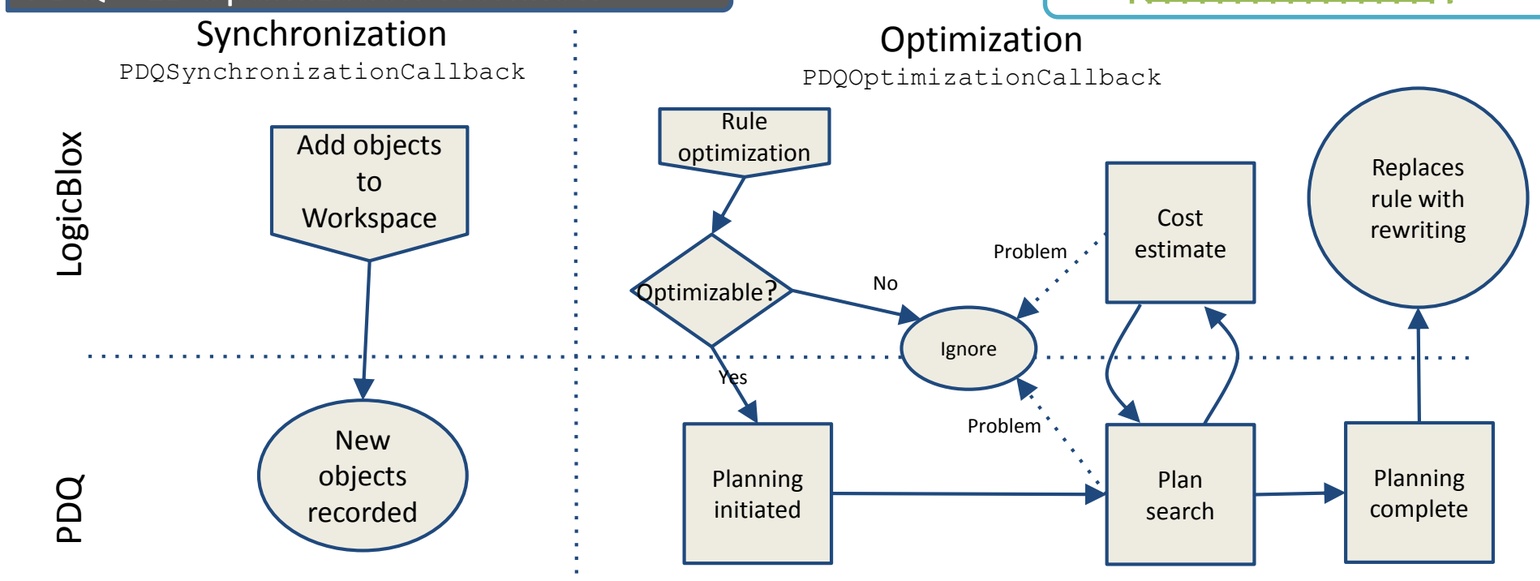
During planning **PDQ asks LB** for its estimation about the **cost** of a specific subplan.

When a transaction ends successfully, PDQ is updated to account for the objects that have been created or destroyed.

LB Architecture (abstract)



PDQ – LB Optimization Workflow



REFERENCES

Benedikt, M., Leblay, J., Cate, B.T. and Tsamoura, E., 2016. Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation. Synthesis Lectures on Data Management, 8(1), pp.1-205.

Benedikt, M., Ten Cate, B. and Tsamoura, E., 2016. Generating Plans from Proofs. ACM Transactions on Database Systems (TODS), 40(4), p.22.

Benedikt, M., Leblay, J. and Tsamoura, E., 2015. Querying with access patterns and integrity constraints. Proceedings of the VLDB Endowment,8(6), pp.690-701.

Benedikt, M., Leblay, J. and Tsamoura, E., 2014. PDQ: Proof-driven query answering over web-based data. Proceedings of the VLDB Endowment,7(13), pp.1553-1556.

Benedikt, M., Ten Cate, B. and Tsamoura, E., 2014, June. Generating low-cost plans from proofs. In Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 200-211). ACM.