SOFSEM 2010
January 23–29, 2010, Špindlerův Mlýn, Czech Republic

Query Optimization through Cached Queries for Object-Oriented Query Language SBQL

Piotr Cybula
Institute of Mathematics and Computer Science, University of Łódź, Poland
<cybula@math.uni.lodz.pl>

Kazimierz Subieta
Institute of Computer Science, Polish Academy of Sciences, Poland
Polish-Japanese Institute of Information Technology, Poland
<subieta@ipipan.waw.pl>

Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Stack-Based Approach

- In SBA a query language is treated as a **special kind of programming language**.
- **Evaluation of queries** is similar to evaluation of expressions in programming languages.
- In the approach each name appearing in the query is bound to a run-time entity according to the scope of its name. Scopes are managed through **Environment Stack** (ENVS).
- Approach can be applied to **arbitrary model of data** (object, object-relational, XML).
- SBA has its **own query language SBQL** (Stack-Based Query Language).

Query Language - SBQL

- In SBQL a query is:
  - Literal, name of variable, function, view e.g. `2, „Smith”, a, Dept`
  - σ q, where q is a query and σ is an unary operator
  - $q_1 \theta\ q_2$ where q_1, q_2 are queries, and θ is a binary operator
- More complex queries are created from the simpler ones.
- Example queries in SBQL:

```
2 + 2, Book, Book.title,
  Book where count(author) > 2
```

- Extension of SBQL: imperative statements (e.g. update, delete, insert, etc.), procedures.

Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Cached Queries

- Redundant data structures stored at database server side (like indices) – **available for all clients**.
- **Automatically created** as a side effect of standard query evaluation process.
- A pair: compiled query syntax tree and **materialized query results** (augmented with additional data for maintenance purposes).
- Cached queries with selection operation only (with operator where) are the **generalization of indices**.
- Similar to materialized views, but the views are manually added and the number of **cached queries could be millions**.

Optimization Method

- Using of cached queries is **totally transparent** for query programmer and his code.
- New query syntax tree is scanned for a subtree equivalent to any cached query syntax tree – found **subtree is replaced with a function call** which returns the cached query results.
- Using of cached results is quick and **independently of query complexity** and actual database state.
- **Query normalization and decomposition** are ways to increase a probability of utilization of the cached results.
- **Automatic cache update** after database modifications is needed for keeping query results up-to-date (only influenced cached queries).

Query Normalization

- Avoidance of placing in the cache two queries syntactically different but with **the same semantics**.
- Query **syntax tree modification** preserving original query result.
- Main **normalization methods**:
  - operand sorting: b<a → a>b
  - operator ordering: a-b+c-d → a+c-b-d
  - auxiliary name unification: Emp **as** e1 → Emp **as** aux1
  - object name unification, for example using only lower case letters in the names of objects

Query Decomposition

- Factoring out **simpler independent queries** for high reuse ability.
- Simpler query is **simply updateable**.
- Main **decomposition methods**:
  - factoring out nested independent subqueries:

    Emp **where** salary < (Emp **where** name = "Smith").salary
  - factoring out aggregation queries (as cached virtual subobjects): Dept **join** avg(employs.Emp.salary)
  - omitting ending path expressions (quickly referenced in object DBs): (Dept **where** count(employs) > 1).boss.Emp.salary
  - query segmentation using set algebra and logical transformations:

    (q **where** p1 **or** p2) → (q **where** p1) ∪ (q **where** p2)

Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

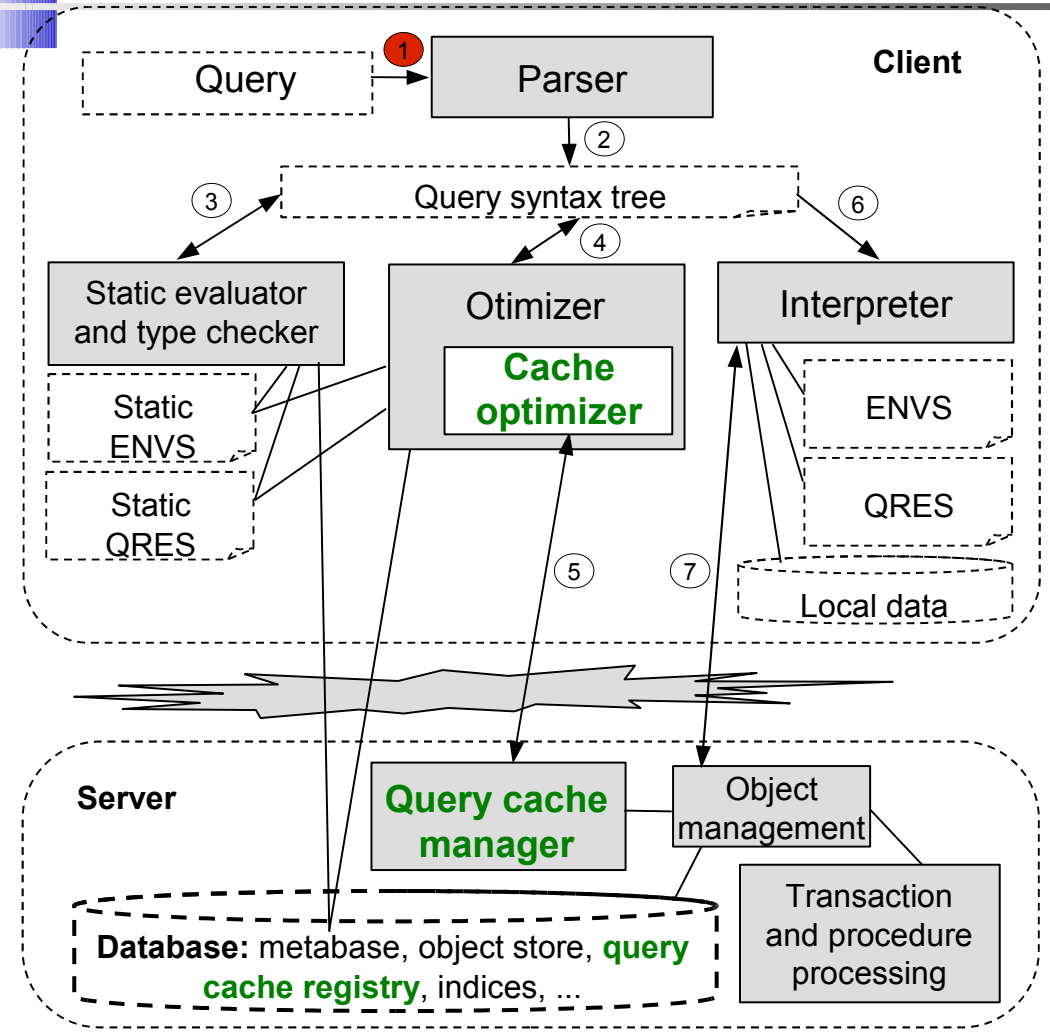
Query Cache Registry

- Cached queries are stored in the cache registry, **uniquely identified** and managed by cache manager.
- Additional **query index** for fast access and search capabilities – linear hashing table with single text key being **normalized text form** of a query.
- Two registry storing places: **physical** (for often used queries) and **volatile** (temporary, for new cached queries).
- Registry parameters configured by database administrator: min. query usage count, max. result size, min. execution time, max. cache size, percent of cache usage size, etc.
- Optimal resource (cache disk space) management using statistics stored for any consecutive cached query – MRU lists maintained for **removing rarely used query results from the cache**.
- **Automatic result update** after database changes.

Outline

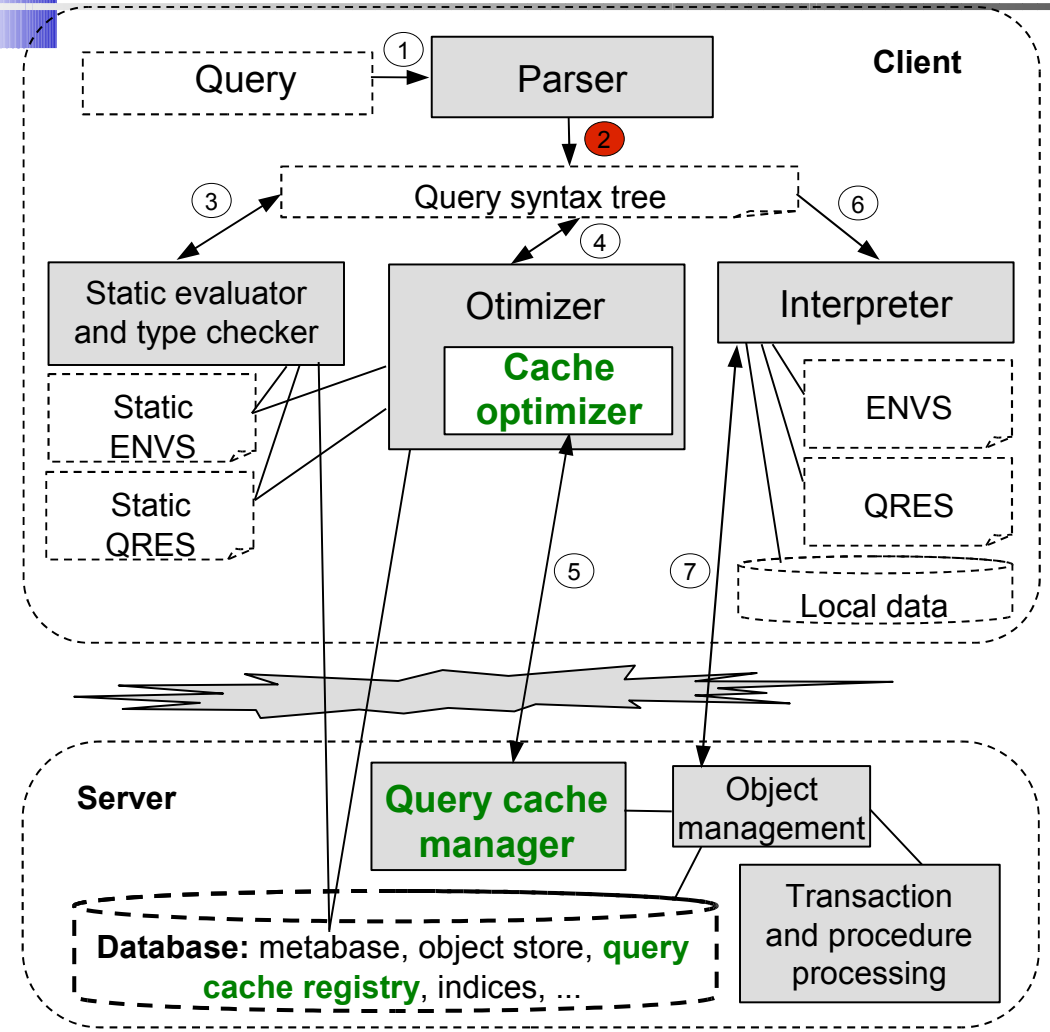
- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Optimizer Architecture



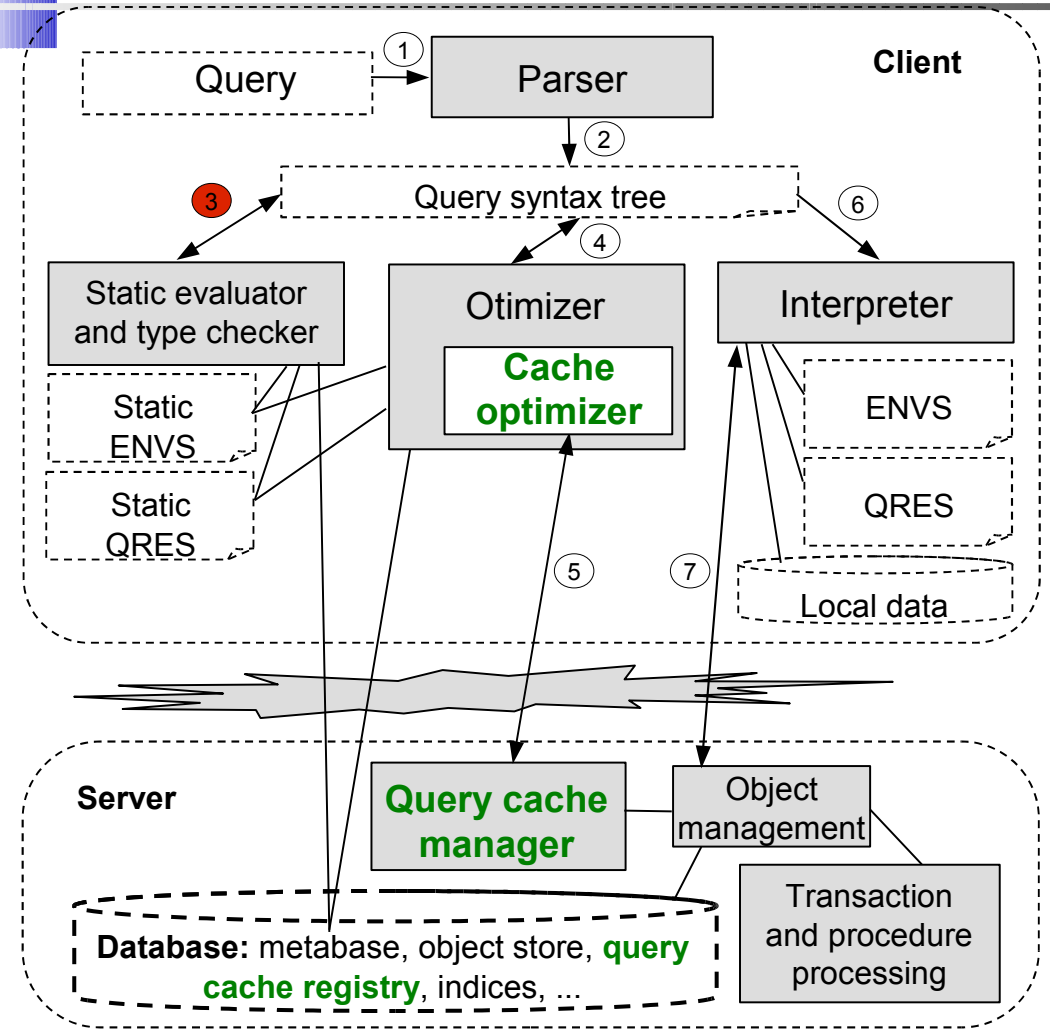
1. Application requests a **query**.

Optimizer Architecture



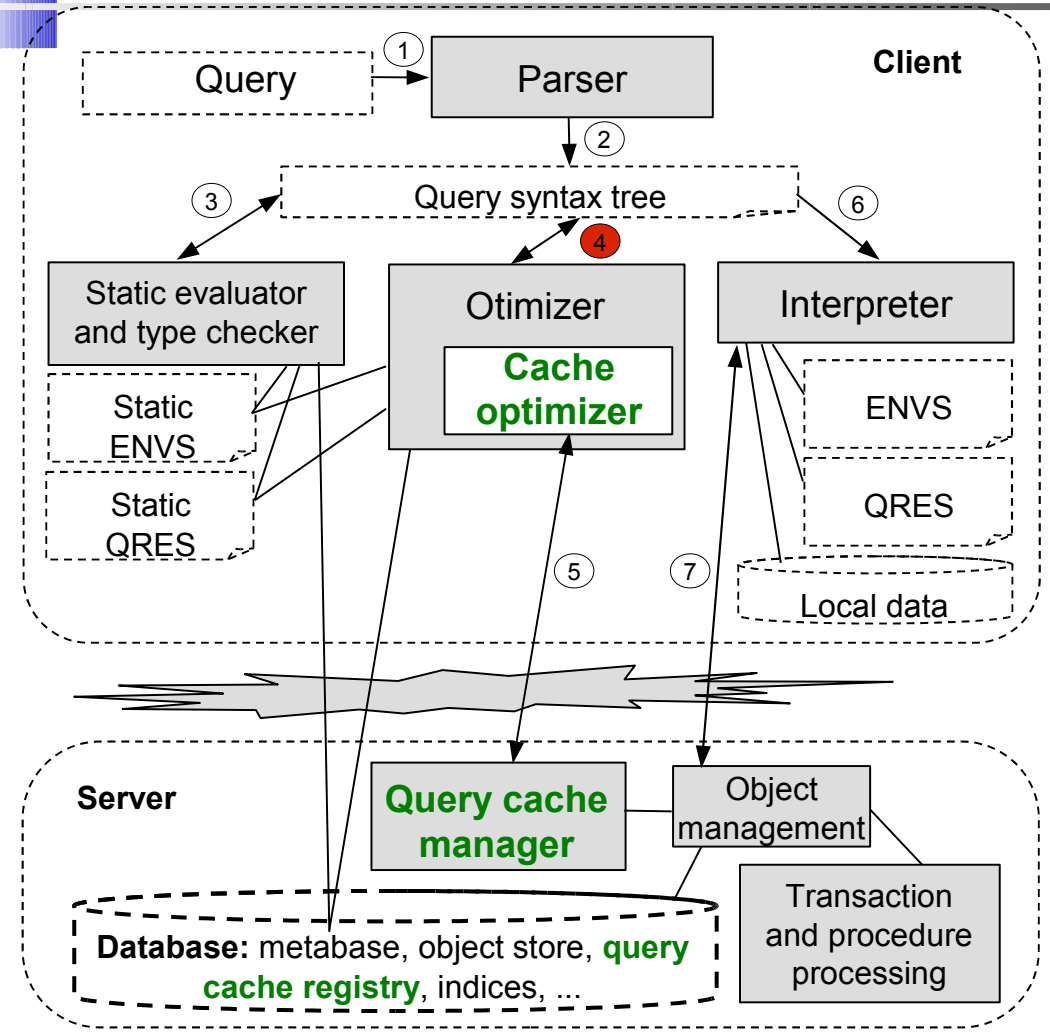
2. **Parser** transforms the query into **query syntax tree**.

Optimizer Architecture



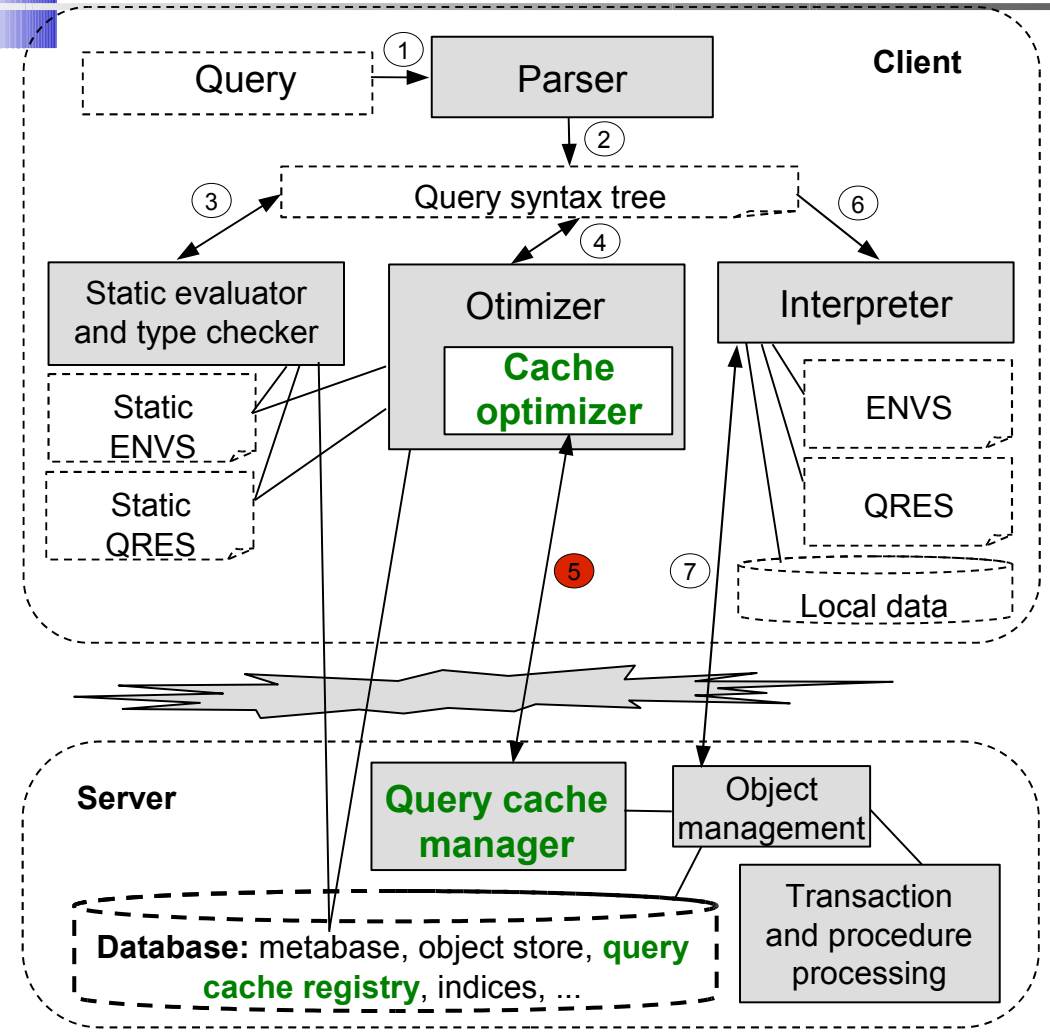
3. The query is **statically evaluated** (using static ENVS, QRES and server-side metabase) for type checking and inserting additional query elements (implicit dereferences, casts, result signatures, etc.).

Optimizer Architecture



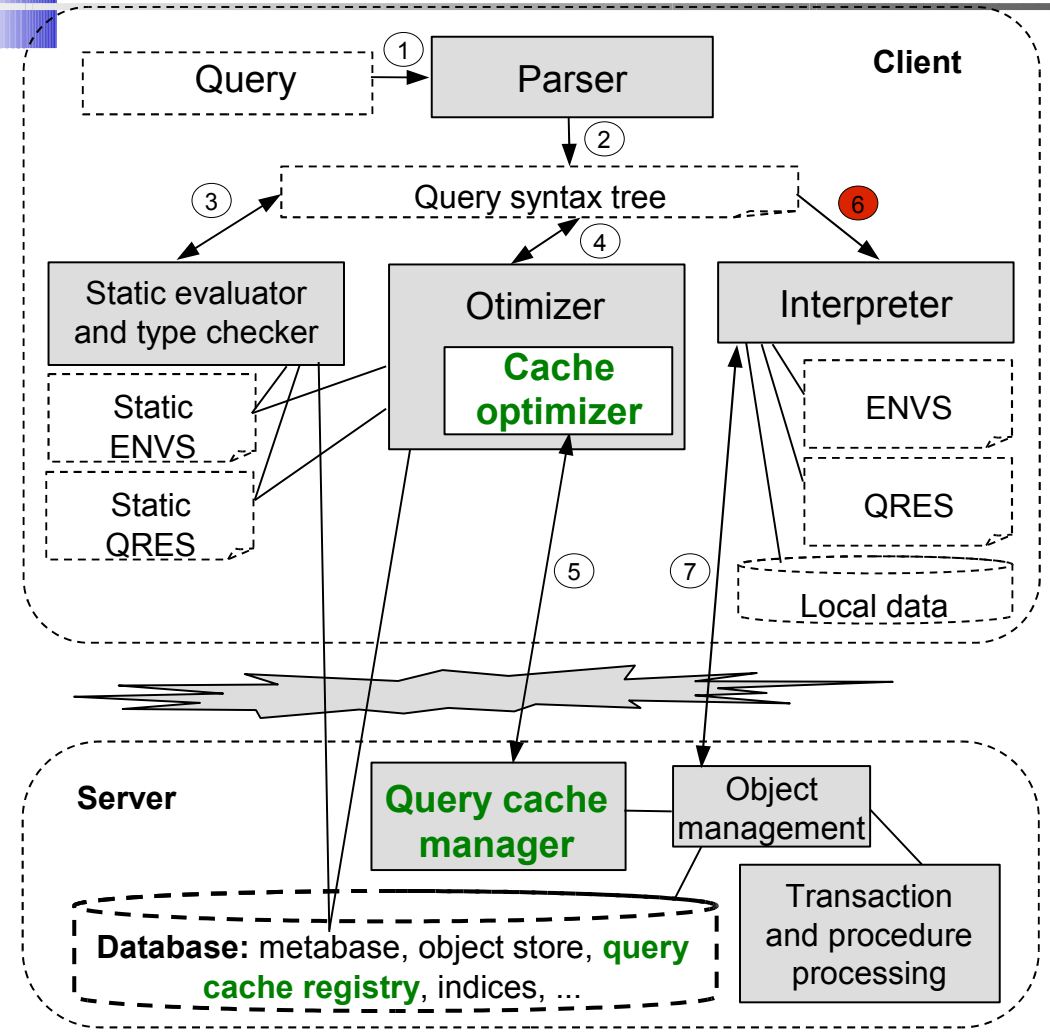
4. The query is sent to optimizer. Query **optimizer is a client-side module**.

Optimizer Architecture



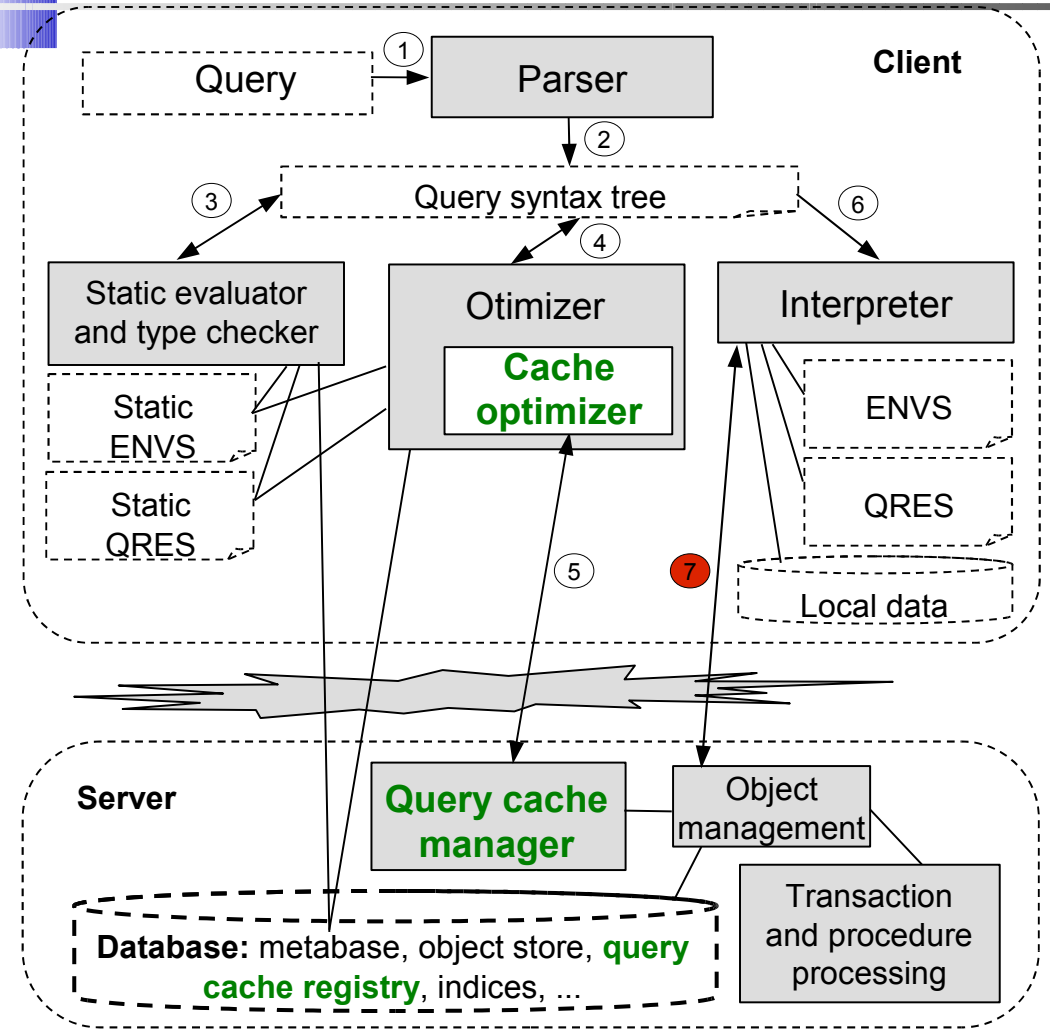
5. Cache optimizer **normalizes and decomposes** the query. Query text form is sent to server-side cache manager for **query matching**. If matched, cache manager **returns query identifier** to the optimizer, else **adds new cached query**. Optimizer modifies query tree by a call of **cache function**.

Optimizer Architecture



6. A **query execution plan** is compiled and sent to the query **interpreter**. Queries are **evaluated on client-side** requesting necessary data from database server.

Optimizer Architecture

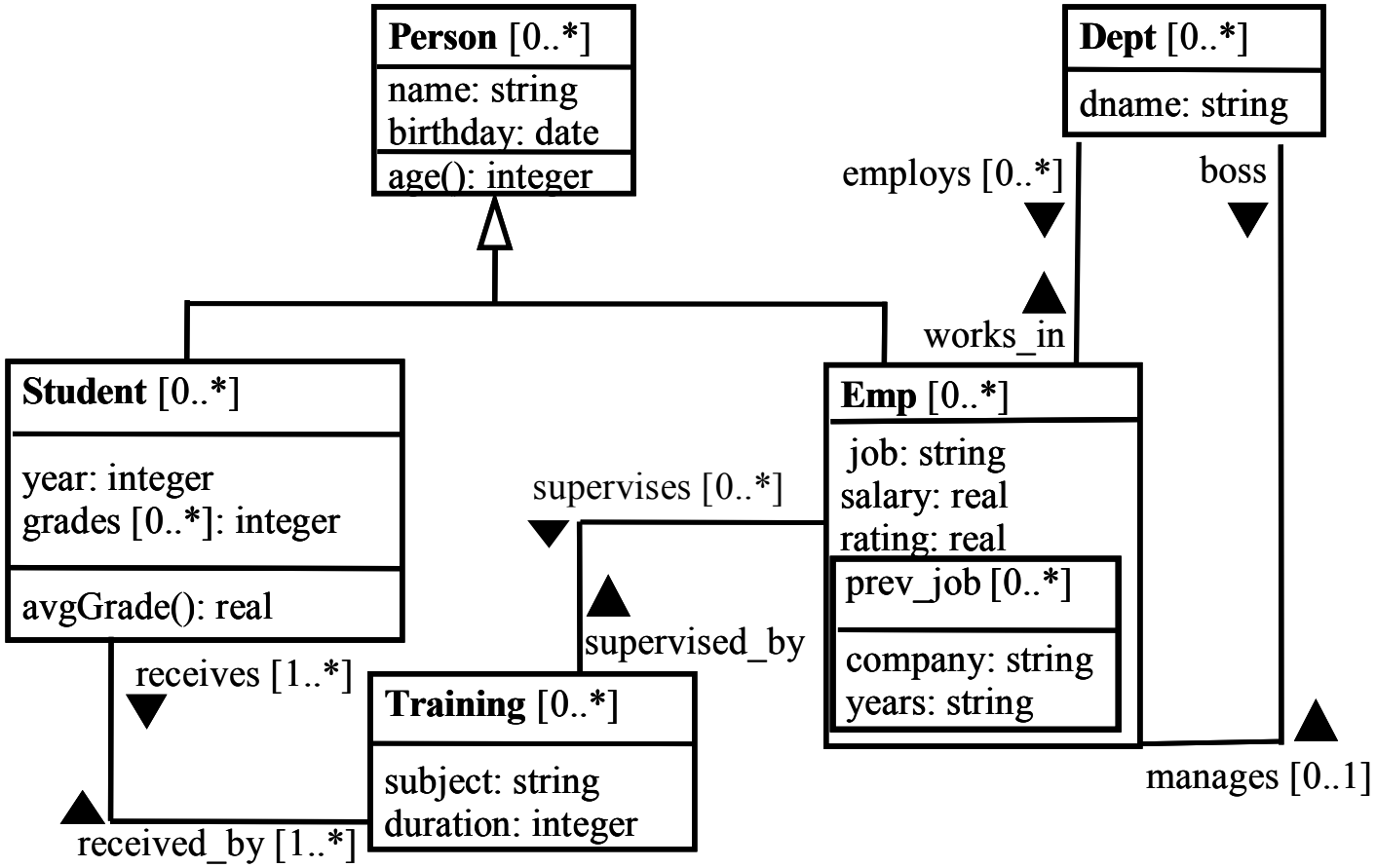


7. Query interpreter evaluates the query. In case it meets a **cache function call**, it requests to server for appropriate **cached query results** (identified by the function parameter value). If the requested cached query is new (without results cached), it **evaluates them and sends** to the cache registry.

Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Exemplary Object Database



Exemplary Optimization Case

- Cached query Q1:

  Emp **join** works_in**.**Dept

- New query:

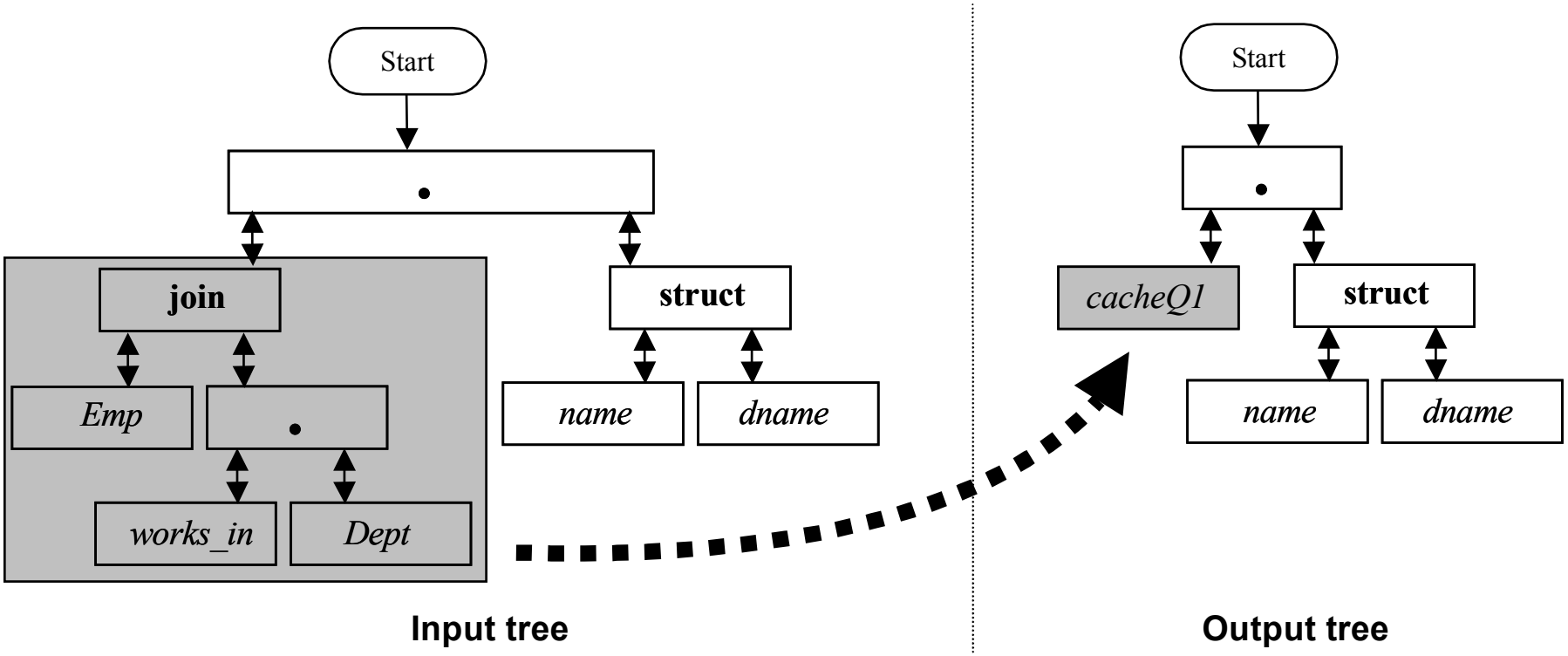
  (Emp **join** works_in**.**Dept)**.**(name, dname)

- After the new query decomposition (cutting out ending path expressions) the cached query could be matched.
- In object DBs path expressions are very quickly evaluated thanks to referential data model.
- New query is modified during optimization process by replacing matched subtree with a cache function call parametrized by a identifier of the cached query Q1:

  CacheQ1**.**(name, dname)

Query Tree Transformation

(*Emp* **join** *works_in.Dept*)**.**(*name, dname*)



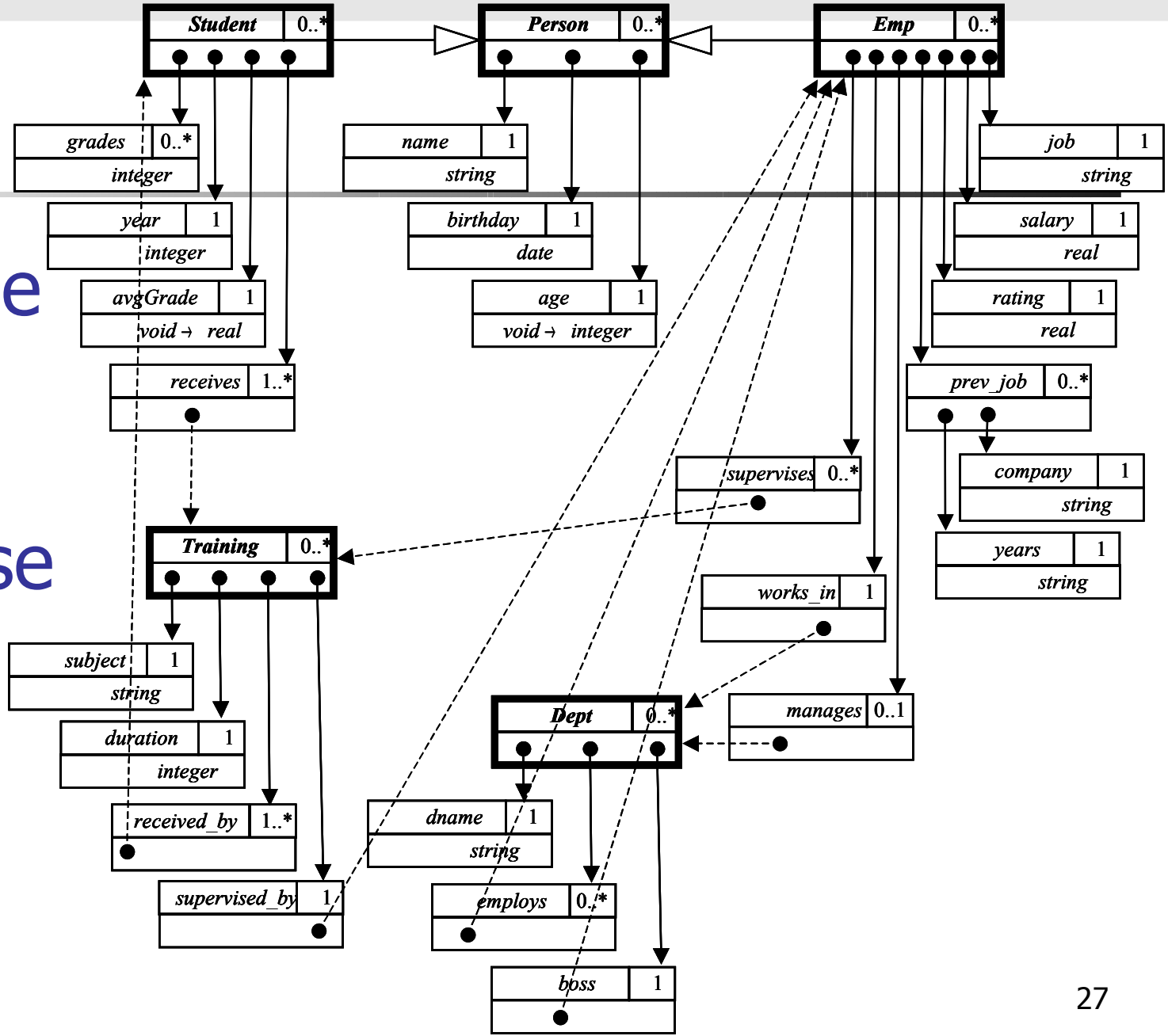
Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Maintenance of Cached Results

- **Immediately** after database modification (using update trigger) or **deferred** (just before next result request or after a group of changes).
- Smart recognition of a subset of cached queries influenced by a database update (others are eliminated – **elimination method** reducing update time).
- Often or with much time consumption updated queries are **removed from the cache registry**.
- Each cached query is stored together with its **sufficient subschema** describing a part of database schema (stored in metabase) necessary to process the query.
- Similar subschemas are generated for each database **update operations** (create, insert, update, delete).

Database schema graph - metabase



Subschema Definitions

- **Subschema function** (subQ) for a query q returns:
  - a set of identifiers of schema nodes containing object names mentioned in query q
- **Subschema function** (subU) for an update u returns:
  - a set of identifiers of schema nodes containing object names influenced by update u (created, inserted, updated or deleted) together with names of their subobjects and superobjects (from base classes)
- Cached queries with **subschemas disjoint** with a subschema generated for the current update are eliminated from cached result correction (when the intersection of subschemas is an empty set).

Elimination Example

- Cached query q: Emp **where** (prev_job.company = "ABC")

  subQ(q) = {Emp, prev_job, company}
- Update u1: **delete** Emp.prev_job **where** company = "ABC"

  subU(u1) = {prev_job, company, years}

  subU(u1) $\cap$ subQ(q) $\neq \emptyset$

  After u1 query q should be corrected.
- Update u2: **delete** (Emp **where** count(prev_job) > 0).
  supervises.Training **where** count(received_by) < 8

  subU(u2) = {Training, subject, duration, received_by, supervised_by, receives, supervises}

  subU(u1) $\cap$ subQ(q) = $\emptyset$

  After u2 the results of query q have been unchanged.

Subschema Index

- Association list in form of the **inverted file for fast subschema comparison** (the fastest search structure for set operations with relative small domain).
- Main structure is a directory of all distinct values of indexed sets (**object name identifiers**) tied with occurrence lists (one list for one identifier).
- An **occurrence list** for a name identifier (sorted and compressed) consists of identifiers of all cached queries with subschemas containing this identifier.
- All queries with identifiers belonging to the occurrence lists tied with name identifiers being elements of subschema sufficient for an update are **designated for correction**.

Incremental Cache Update

- Results of cached queries (not eliminated after database update occurred) are **reevaluated** or **corrected incrementally** by removing such **subresults**, which were generated from modified part of a database and evaluating the results again on that part of data.
- Some kinds of queries cannot be updated incrementally, such as **aggregates or nested queries** (most of them are decomposed by the optimizer).
- Storing query **results divided into parts** derived from different root objects (sum of all the parts is the result).
- Removing results derived from **root objects with access to deleted or updated data** and then, reevaluating a query on database with **root objects with access to new or updated data**.

Outline

- Stack-Based Approach
- Concept of Cached Queries
- Query Cache Registry
- Architecture of Optimizer for SBA
- Query Optimization Case Study
- Automatic Cache Update
- Conclusions and Future Work

Conclusions and Future Work

- **Query result caching** can be used to **enhance performance** of query processing.
- Optimization solutions are **implemented for SBQL language** into **ODRA** (Object Database for Rapid Application development) project of SBA OODBMS built from scratch: (http://sbql.pl/various/ODRA/ODRA_manual.html)
- Experiments showed that in many cases, specially for complex queries, responses were over **100 times faster**.
- **Future research** on result caching will include:
  - taking into account some additional SBA and SBQL features, such as object roles and updatable object views
  - methods for reusing of some parts of cached results or results of many cached queries combined together
  - extending caching solutions for distributed environment and virtual grid database repositories

Thank You for your attention!