A genetic approach for random testing of database systems

Hardik Bati, Leo Giakoumakis, Steve Herbert, Aleksandras Surna

Microsoft Corporation

Motivation

- Random testing techniques have been proved to be useful for testing large, complex software systems
- The use of random testing in SQL Server has been valuable for several product releases
- Particularly the use of the RAGS system: *Slutz, D. Massive Stochastic Testing of SQL, In Proceedings of the 24th VLDB Conference, (New York USA 1998), 618-622*

Challenges

- Query processor testing challenges:
  - Practically infinite input space
  - Dynamic code paths
  - Difficult to test in isolation
- Random testing challenges:
  - Ensuring that random tests hit desired targets
  - Directing the generation process towards desired targets
- RAGS limitations:
  - Generated queries often contain logical contradictions
  - Most complex queries don’t return results

Outline

- Random testing in SQL Server
- The genetic approach to random testing
- Experimental results

Random testing in SQL Server

- An integral part of our testing process
- Used in parallel with other testing methods
- Random testing has been invaluable:
  - Particularly useful during big code restructuring efforts
  - Non-trivial defects are found earlier in the test development cycle
  - Inexpensive way to build very complex test cases

History of Random testing in SQL Server

- Query compiler architecture changed during the 2000 release
  - Used the RAGS tool developed by Microsoft Research
  - Made several extensions since the original version
  - Uncovered a large number of defects

- SQL server 2005 included significant changes in the query processor and many new features
  - Used the method presented in the paper in parallel with RAGS
  - The new method discovered 10 times more defects

Example defect in SQL 2005

```
SELECT soundex(_s4_) _s0_ ,   atan(_n5_) _n1_ ,
dbo.ufnGetProductStandardCost(_n5_, _d6_) _o2_
from     (
   select [JobCandidateID] _o8_, [Edu.StartDate] _d7_,
   [Edu.EndDate] _d6_, [Edu.Major] _s9_, [Edu.Minor] _s4_,
   [Edu.GPA] _s10_, [Edu.GPAScale] _s11_, [Edu.School] _s12_,
   [Edu.Loc.CountryRegion] _s13_, [Edu.Loc.State] _s14_,
   [Edu.Loc.City] _s15_, Edu.Major] _n16_ ,[ContactID] _n5_ ,
HumanResources.[vJobCandidateEducation]
OUTER APPLY dbo.ufnGetContactInformation([Edu.Major]) as
TVF1) t0
option (loop join)
```

2. XML column

- All three elements had been tested independently
- The specific combination of all three was not
- The defect was found by a customer 2 months later

3. Parallel plan

1. table-value function

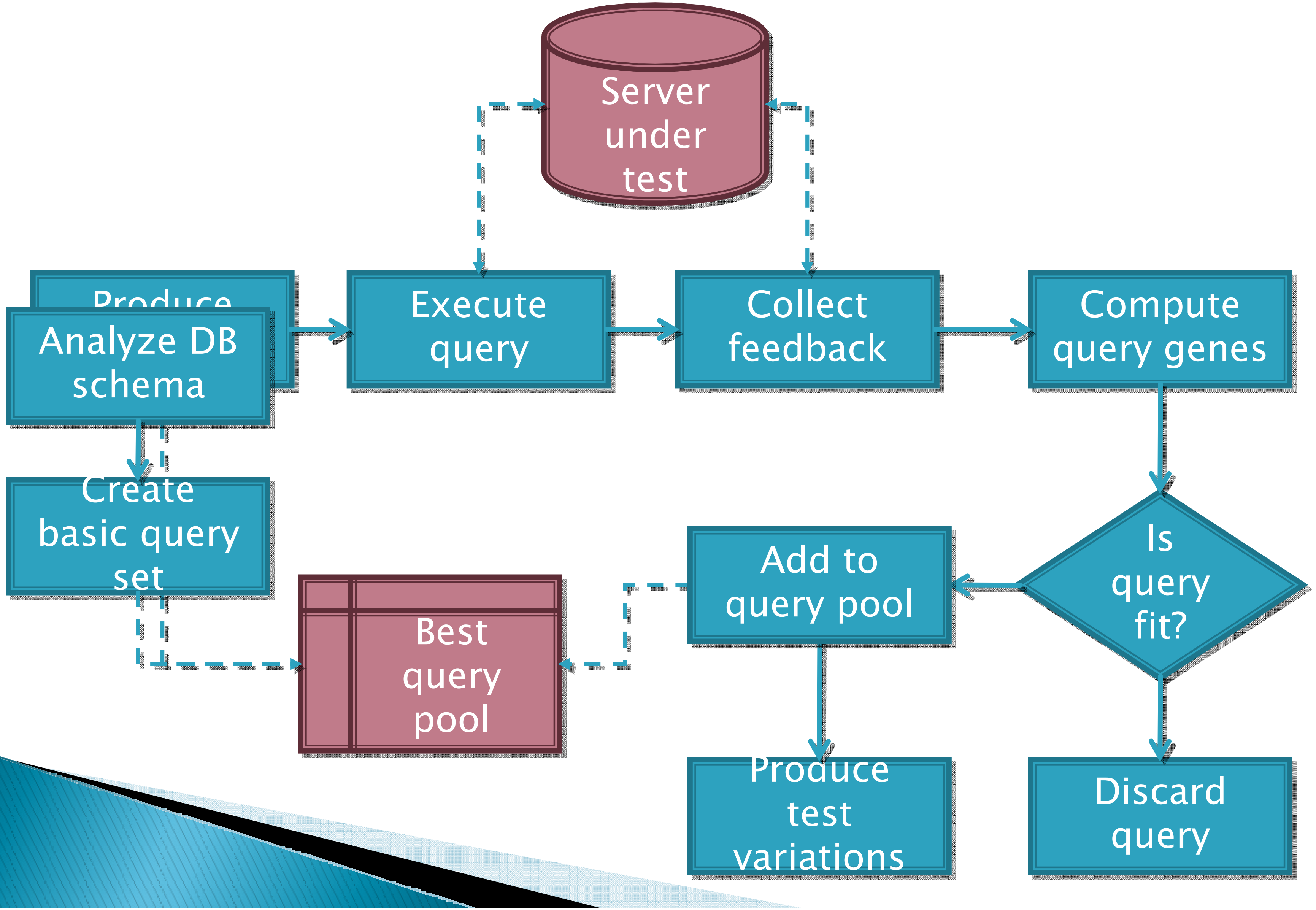
Outline

- Random testing in SQL Server
- The genetic approach to random testing
- Experimental results

Method

- A simple genetic algorithm produces SQL queries by combining or mutating existing ones
- The genetic process is guided by feedback from query execution against the server under test
- Execution feedback is represented as *query genes*
- The algorithm tries to produce new queries with unique gene combinations
- Defects are found by the self-checking mechanisms of the server (asserts) and by comparing results with a trusted/previous version of the server

Test generation process



SQL Query reproduction

- New queries are produced by mutating or combining one or more queries from the *best query pool*

- Query synthesis techniques are enabled by the composability of SQL language

- The paper describes a variety of synthesis techniques; here we present only some basic examples

Query synthesis using JOIN

```
SELECT _s12_ _s13_ ,_n14_ + _n14_ _n15_
FROM
(
   SELECT [L_ORDERKEY] _n16_, [L_PARTKEY]
   _n17_, [L_EXTENDEDPRICE] _n18_, [L_DISCOUNT]
   _n19_, [L_TAX] _n20_, [L_RETURNFLAG] _s21_
   FROM tpch100m.dbo.[LINEITEM]
) t0 RIGHT OUTER JOIN (
   SELECT [O_TOTALPRICE] _n14_, [O_COMMENT]
   _s12_
   FROM tpch100m.dbo.[ORDERS]
   ) t1 ON _s12_ > _s21_ and _n14_ = _n16_
```

- A new query is created as a JOIN of two basic queries

Query mutation

```
    SELECT max(tt._s12_)
    FROM
         (
             SELECT [O_TOTALPRICE] _n14_,
             [O_COMMENT] _s12_
       FROM tpch100m.dbo.[ORDERS]
) tt
```

- A basic query is mutated as a derived table with an aggregate

Query synthesis using sub-query

```
SELECT _s12_ _s13_ ,_n14_ + _n14_ _n15_
FROM
(
  SELECT [L_ORDERKEY] _n16_, [L_PARTKEY] [...]
  FROM tpch100m.dbo.[LINEITEM]
) t0 RIGHT OUTER JOIN (
  SELECT [O_TOTALPRICE] _n14_, [O_COMMENT]   _s12_
  FROM tpch100m.dbo.[ORDERS]
  ) t1 ON _s12_ > _s21_ and _n14_ = _n16_
WHERE _s12_ in
   (
   SELECT max(tt._s12_)
   FROM  (
        SELECT [O_TOTALPRICE] _n14_, [O_COMMENT] _s12_
     FROM tpch100m.dbo.[ORDERS]) tt
     WHERE tt._n14_ = t1._n14_
      )
```

- Combination of the two previous queries as sub-query with correlation

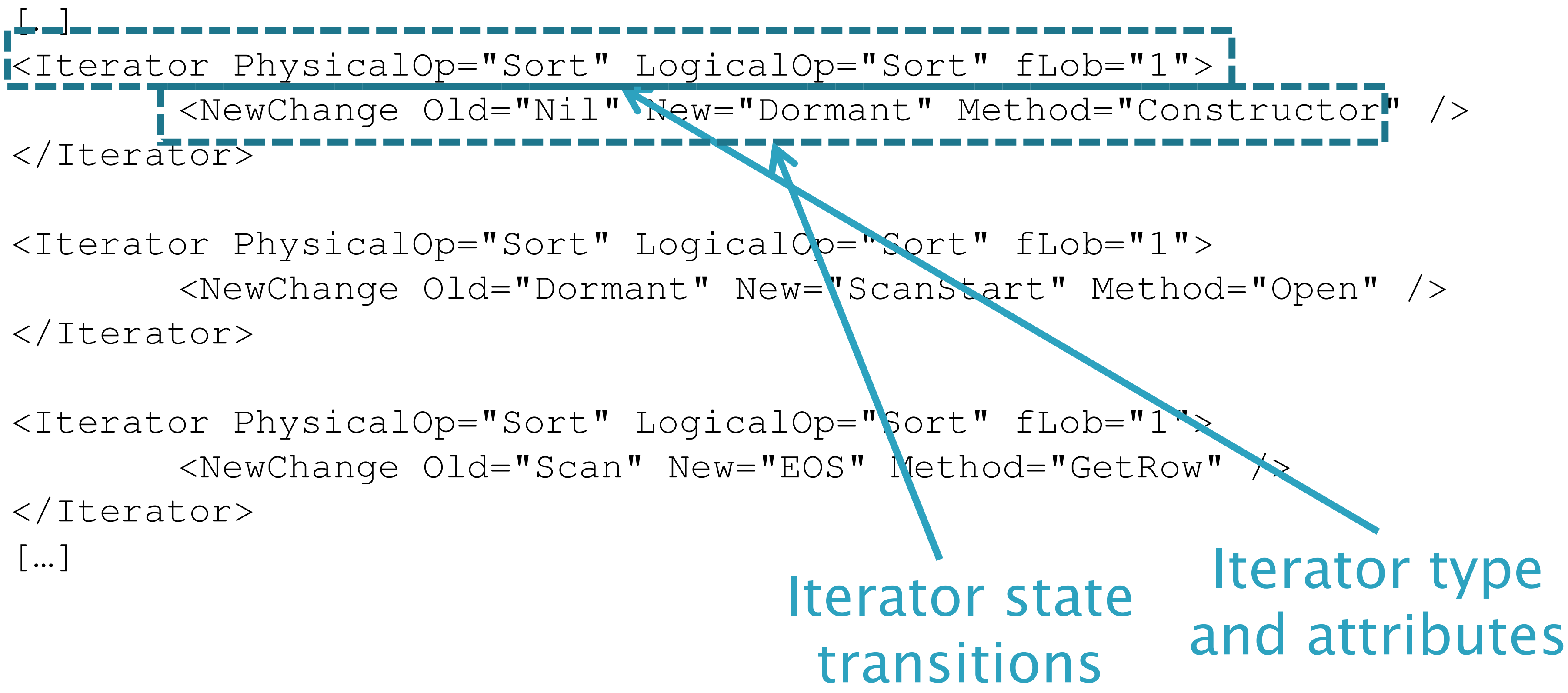
Feedback and query genes

- Genes are based on execution feedback
  - Execution plan
  - Trace information provided by the server

- Query genes describe code coverage:
  - Interesting code paths exercised
  - The context under which those code paths are exercised

- Examples of genes:
  - *“ exercised the [Left Outer Join to Nested Loops] optimization rule”*
  - *“exercised hash join operator” + “parallel query plan”*
  - *“line 555 in source file [hash.cpp]”*.

Example: iterator coverage feedback



```
[…]
<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
        <NewChange Old="Nil" New="Dormant" Method="Constructor" />
</Iterator>

<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
        <NewChange Old="Dormant" New="ScanStart" Method="Open" />
</Iterator>

<Iterator PhysicalOp="Sort" LogicalOp="Sort" fLob="1">
        <NewChange Old="Scan" New="EOS" Method="GetRow" />
</Iterator>
[…]
```

- Execution feedback provided by the server in the form of XML trace
- Describes iterator types and their state transitions

Example: Optimization rules coverage feedback

Rule	**Succeeded**
Join to Nested Loops	3
Left Outer Join to Nested Loops	2
Left Semi-Join to Nested Loops	1
Left Anti-Semi-Join to Nested Loops	0
Join to Hash Join	1
Full Outer Join to Hash Join	0

- Execution feedback is provided by the server via a system table.
- It describes the set of optimization rules exercised

Query fitness

- The genetic process remembers the set of genes of each query and its frequency
- During the reproduction process queries with rare genes are preferred
- New queries with genes seen for the first time are added to the *best query pool*
- New queries with genes that were seen before, are still added to the pool
  - If they are more readable
  - Execute faster

Outline

- Random testing in SQL Server
- The genetic approach to random testing
- Experimental results

Evaluation

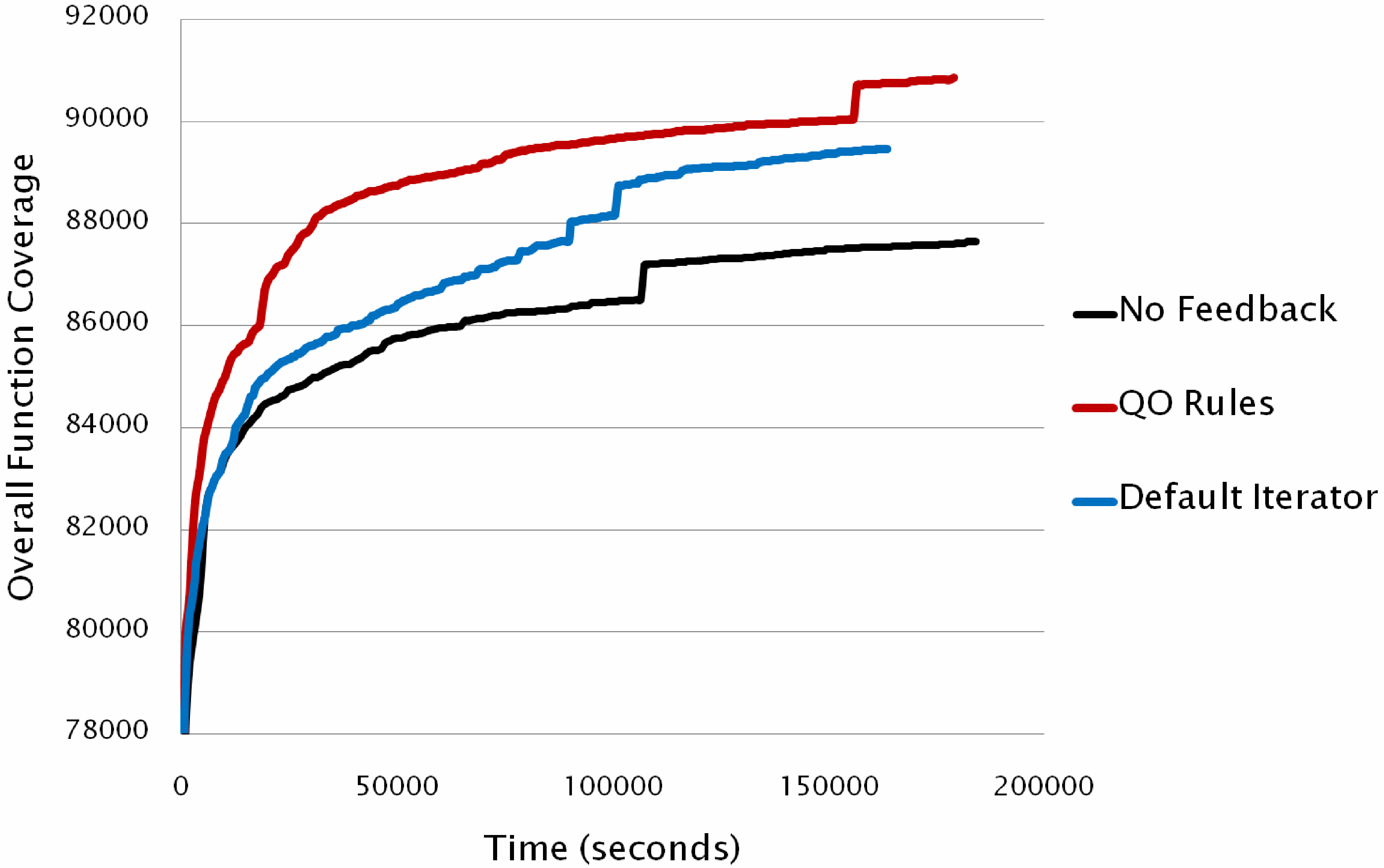
- We present results from three different experiments:
  - With feedback describing optimization rule coverage
  - With feedback describing iterator coverage
  - Without special feedback

- We also compare results with RAGS

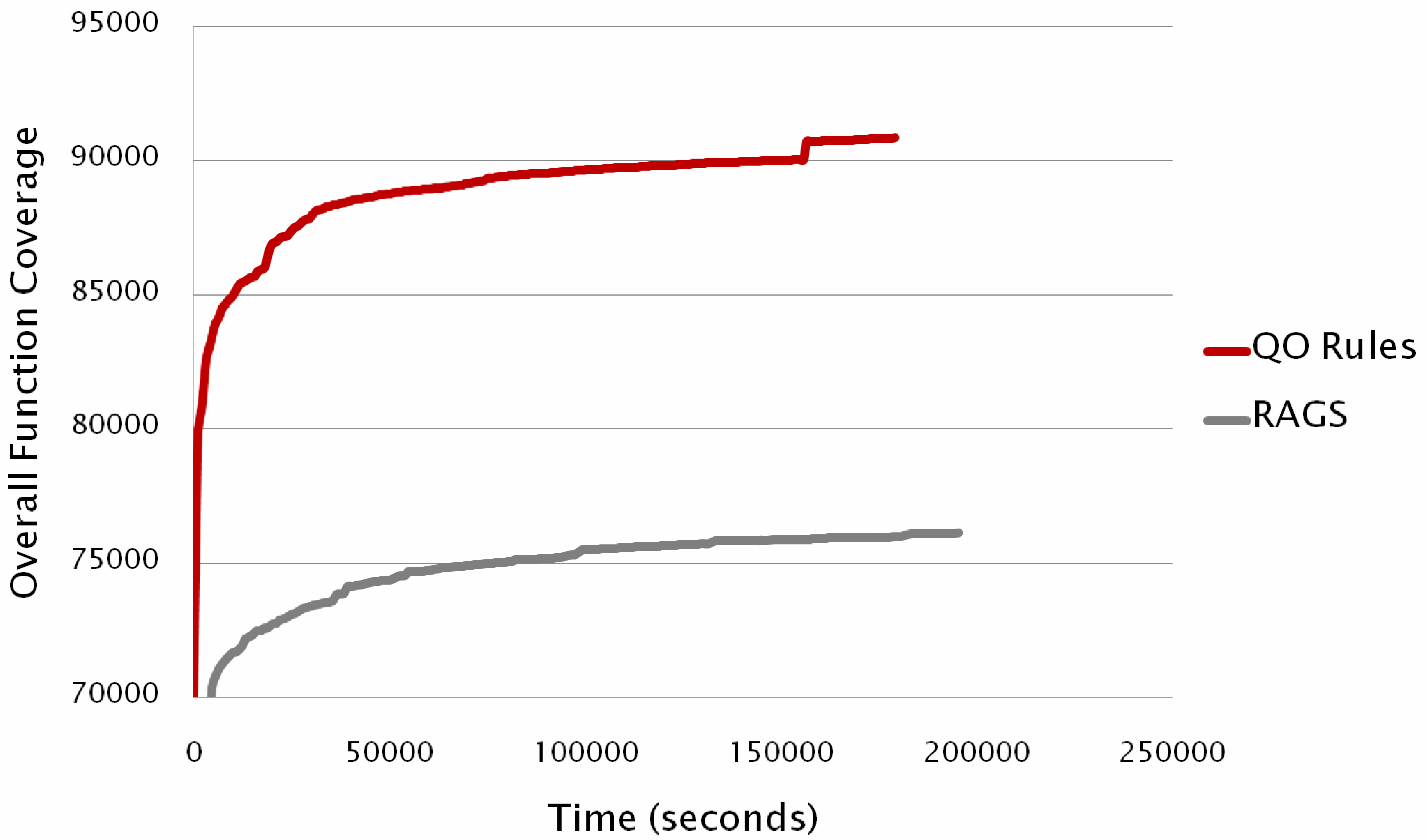
- Experiments were done:
  - on a pre-release version of SQL Server 2008
  - using a database from TPC-H
  - over a period of 48 hours

- Code coverage was measured in unique function invocations (function, function-caller pairs)

Different feedback strategies



Genetic method vs. RAGS



Summary

- We discussed how random testing is used in SQL Server
- We presented a new practical technique for random test case generation, which outperforms previous methods
- We showed that the use of different types of execution feedback improves the effectiveness of random testing

Questions?