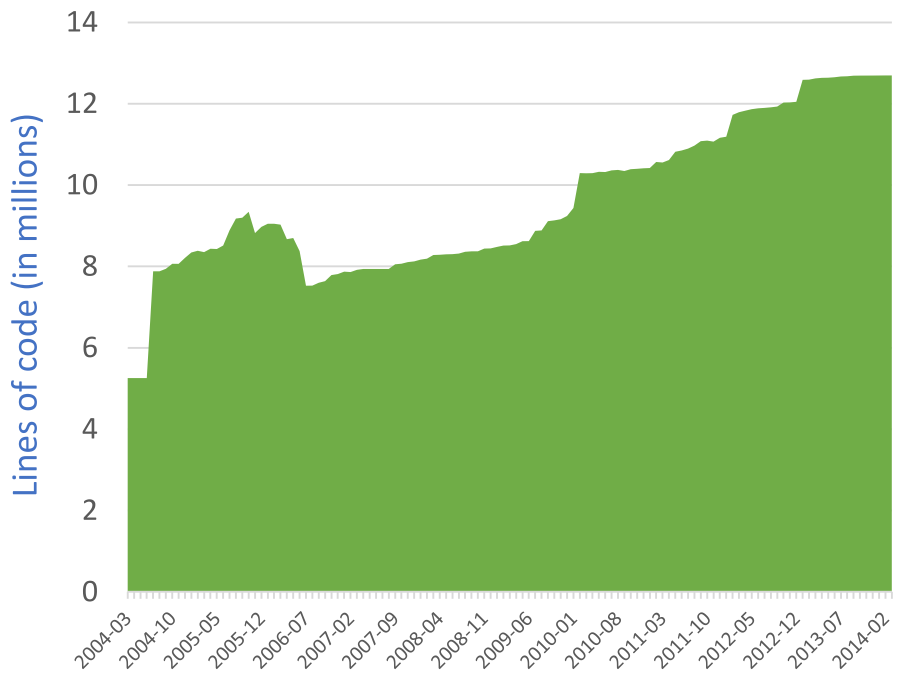
Performance Regression Testing Target Prioritization via Performance Risk Analysis

Peng Huang, Xiao Ma, Dongcai Shen, Yuanyuan Zhou

University of California, San Diego
University of Illinois at Urbana-Champaign

http://cseweb.ucsd.edu/~peh003/perfscope

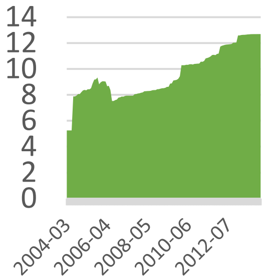
Trend #1: Software evolving fast



*: data from
www.ohloh.net

Lines of code for MySQL over past 10 years grew from ~5 million to ~13 million!

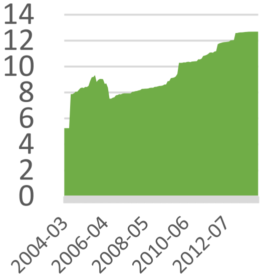
Trend #1: Software evolving fast



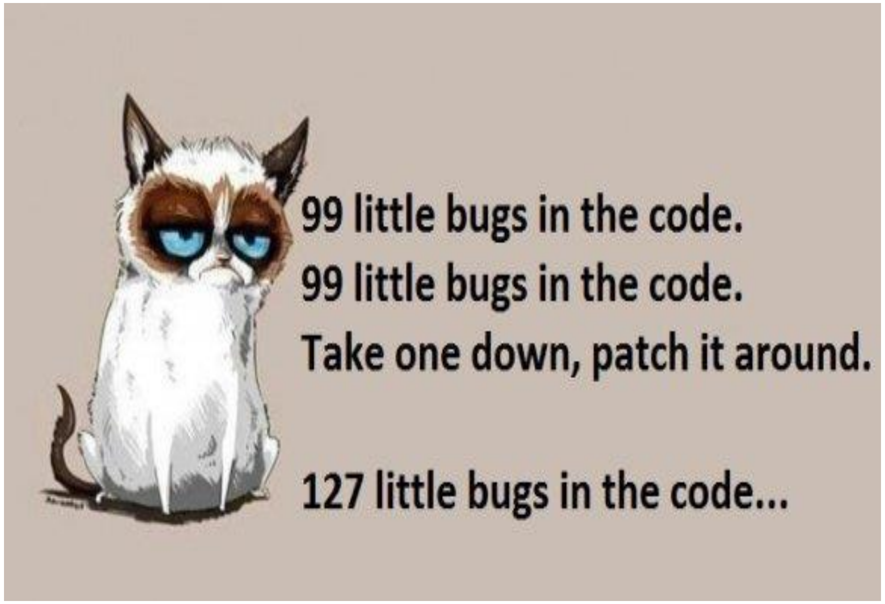
Software	Avg. Rev. Per Day
MySQL	~27
Chrome	~155
Linux	~170

The average revision rate can be ≥ 100 commits per day!

Trend #1: Software evolving fast



Softwar e	Avg. Rev. Per Day
MySQL	~27
Chrome	~155
Linux	~170


99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.

127 little bugs in the code...

Broken functionality or worse performance!

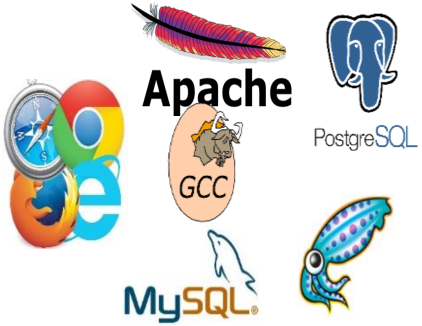
Trend #2: Performance testing, important but slow…

Upgrading MySQL **4.1** to **5.0** in a production e-commerce website:

> “ Although this is a performance issue, total page rendering time in my web shop would increase from **1 second to 20 seconds** for example if showing a decent amount of products and prices on the same page. Therefore MySQL 5 is no good for production until this bug is fixed. ”

Performance critical software

Trend #2: Performance testing, important but slow…



Performance critical software

Category	Test Suite
Web Server	autobench,Web Polygraph,SPECweb
Database	pgbench,sysbench,DBT2
Compiler	CP2K,Polyhedron,SPEC CPU
OS	lmbench,Phoronix Test Suite

Performance regression testing benchmark

Trend #2: Performance testing, important but slow…

Category	Test Suite	Per Run Cost
Web Server	autobench,Web Polygraph,SPECweb	3min—1hr
Database	pgbench,sysbench,DBT2	10min—3hrs
Compiler	CP2K,Polyhedron,SPEC CPU	1hr—20hrs
OS	lmbench,Phoronix Test Suite	2hrs—24hrs

Performance regression testing cost

Problem: Catch me (perf. regression) if you can!

> Doing [performance testing] between just release kernels means that there will be a two-month lag between telling developers that something pissed up performance. Doing it every day (or at least a couple of times a week) will be much more interesting. [. . .] Two months (or half a year) later, and we have absolutely no idea what might have caused a regression. For example, that 2.6.2->2.6.8 change obviously makes pretty much any developer just go : I've got no clue.
>
> -- Linus Torvalds

Current practices of perf. regression testing

- Aggregate testing
  - Daily, weekly, per-release
- Prioritize test cases
  - Divide based on comprehensiveness and overhead
  - Multiple levels

Our tool—*PerfScope*—in a nutshell

- Prioritize perf. regression testing *target* with *Performance Risk Analysis*
  - Statically examine a code commit
  - Conduct performance risk analysis
- Lightweight, white-box
- NOT a performance bug detection tool

Our tool—*PerfScope*—in a nutshell

- Prioritize perf. regression testing *target* with *Performance Risk Analysis*
  - Statically examine a code commit
  - Conduct performance risk analysis
- Lightweight, white-box
- NOT a performance bug detection tool

Outline

- Understanding real world performance regression issues
- Performance risk analysis design
- Implementation: PerfScope
- Evaluation
- Conclusion

Performance regression study

- What do real world performance regression issues look like?
- Is there opportunity to statically analyze the performance impact of code change?
- If so, based on the real world issues, what static analysis is needed?

Study subjects

Software	Description	# of issues
MySQL	DBMS	50
PostgreSQL	DBMS	25
Chrome	Web Browser	25

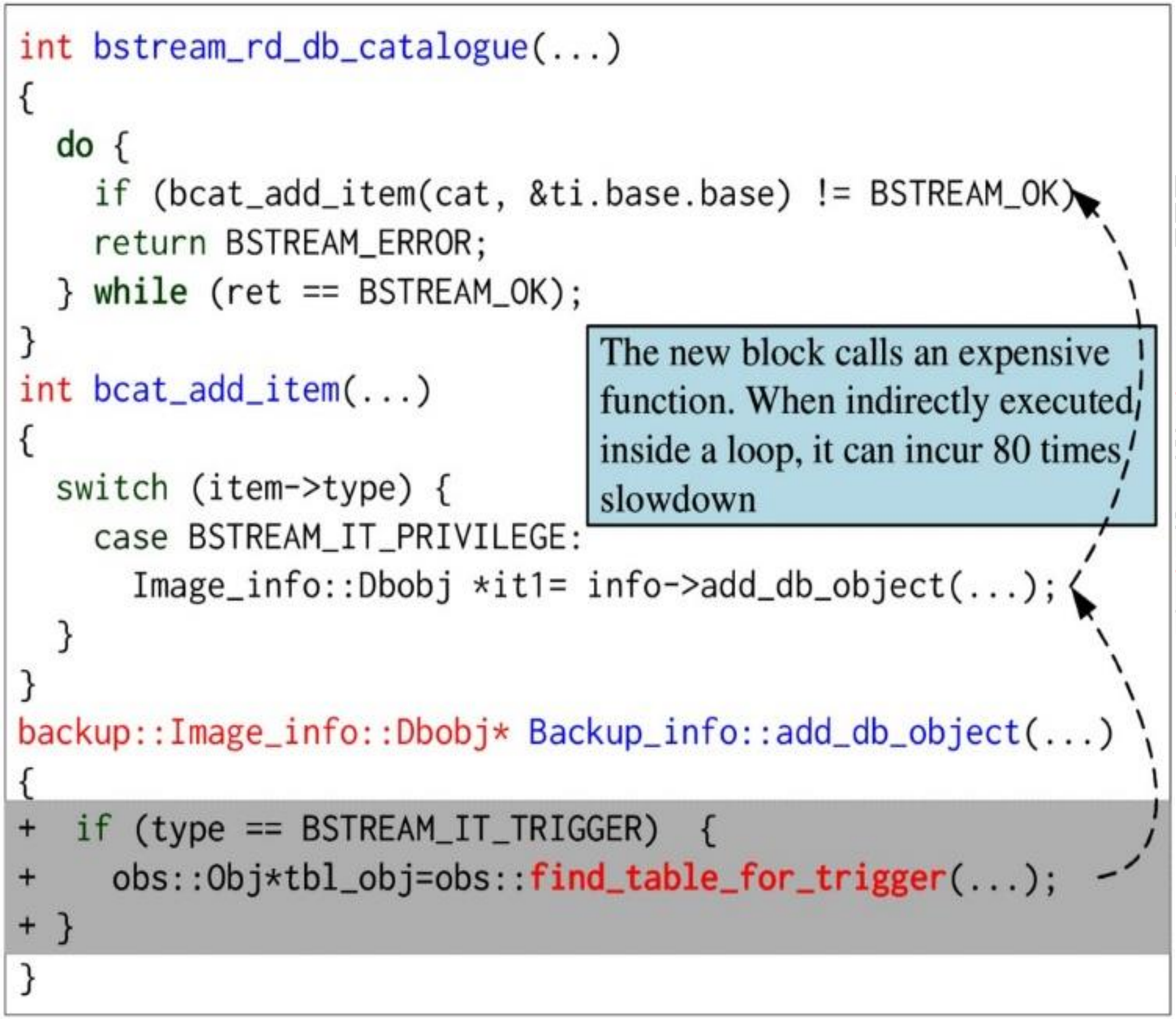
Studied software of real world performance regression issues

Categorizing problematic code changes

```
Foo()
{
    …
    do_add;
    do_add;
    do_add;
    …
}
```

Performance	Performance regression
Cost of **`do_add`**	↑
Execution frequency of **`do_add`**	↑

Where the problematic change takes place?



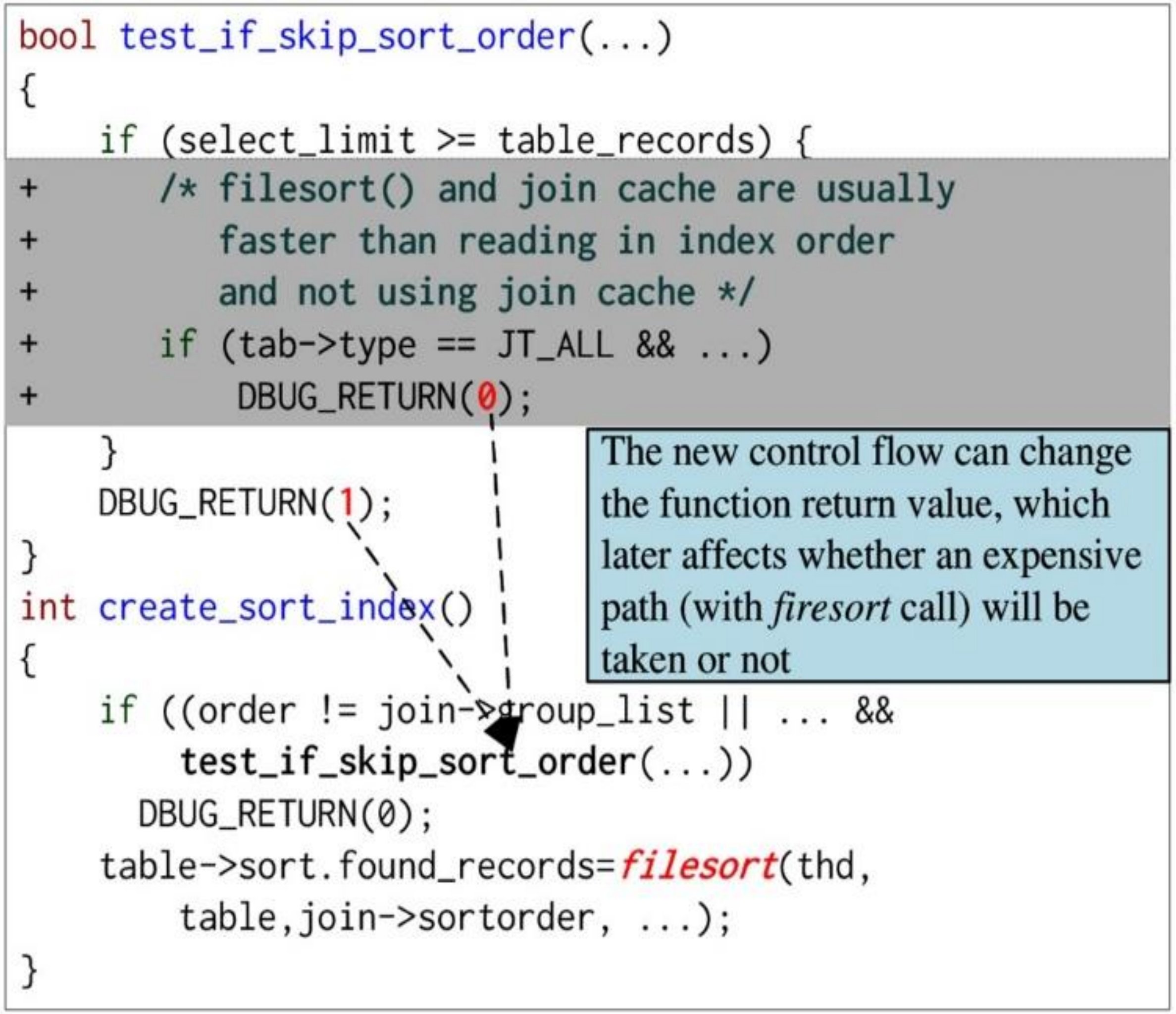
```
int bstream_rd_db_catalogue(...)
{
  do {
    if (bcat_add_item(cat, &ti.base.base) != BSTREAM_OK)
    return BSTREAM_ERROR;
  } while (ret == BSTREAM_OK);
}
int bcat_add_item(...)
{
  switch (item->type) {
    case BSTREAM_IT_PRIVILEGE:
      Image_info::Dbobj *it1= info->add_db_object(...);
  }
}
backup::Image_info::Dbobj* Backup_info::add_db_object(...)
{
+  if (type == BSTREAM_IT_TRIGGER)  {
+    obs::Obj*tbl_obj=obs::find_table_for_trigger(...);
+ }
}
```

The new block calls an expensive function. When indirectly executed inside a loop, it can incur 80 times slowdown

Location	Chrome
API (e.g., Ex	1 (4%)
Utility functi	1 (4%)
Routine func	3 (32%)
Loop	4 (16%)
Others	1 (44%)

Where the problematic change takes place?

Locatio	Chrome
API (e.g., Ex	1 (4%)
Utility functi	1 (4%)
Routine func	3 (32%)
Loop	4 (16%)
Others	1 (44%)



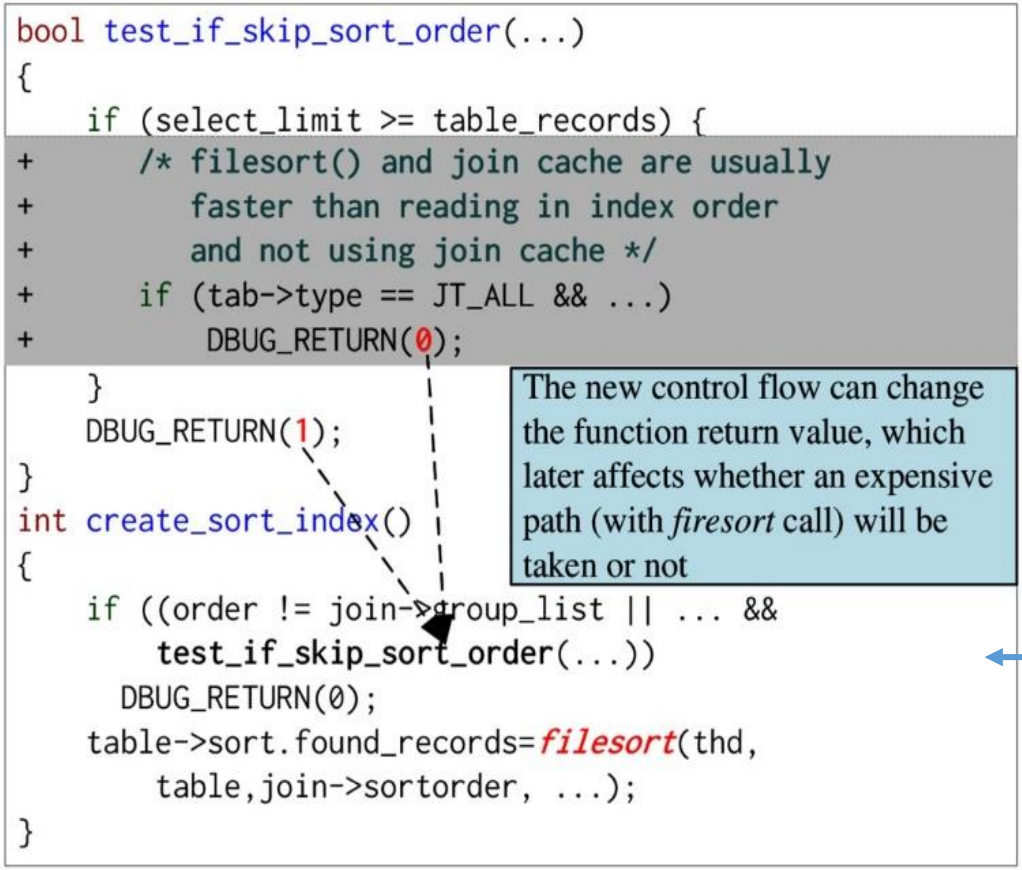
```
bool test_if_skip_sort_order(...)
{
    if (select_limit >= table_records) {
+      /* filesort() and join cache are usually
+         faster than reading in index order
+         and not using join cache */
+      if (tab->type == JT_ALL && ...)
+          DBUG_RETURN(0);
    }
    DBUG_RETURN(1);
}
int create_sort_index()
{
    if ((order != join->group_list || ... &&
        test_if_skip_sort_order(...))
      DBUG_RETURN(0);
    table->sort.found_records=filesort(thd,
        table,join->sortorder, ...);
}
```

The new control flow can change the function return value, which later affects whether an expensive path (with *firesort* call) will be taken or not

What the problematic change modifies?

Modified program elements	MySQL	PostgreSQL	Chrome
Expensive function call	21 (42%)	9 (36%)	16 (64%)
Performance sensitive condition	8 (16%)	6 (24%)	4 (16%)
Performance critical variable	6 (12%)	5 (20%)	2 (8%)
Others	15 (30%)	5 (20%)	3 (12%)

What the problematic change modifies?



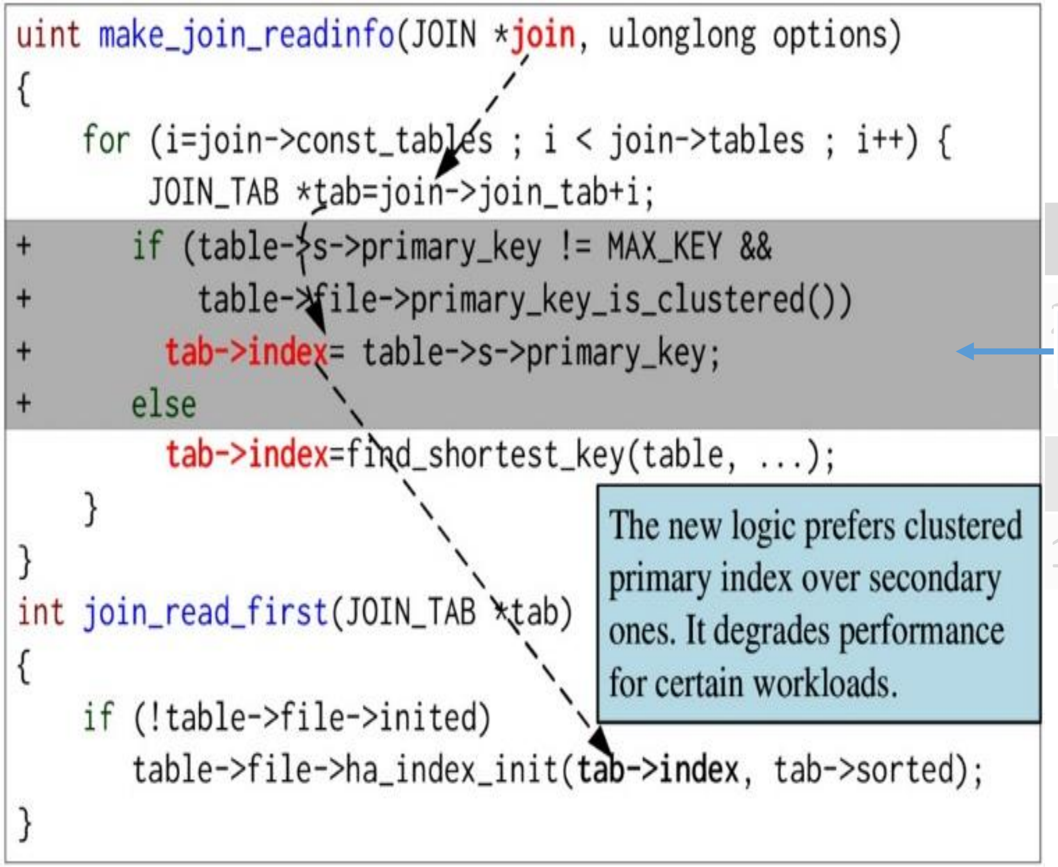
```
bool test_if_skip_sort_order(...)
{
    if (select_limit >= table_records) {
+      /* filesort() and join cache are usually
+         faster than reading in index order
+         and not using join cache */
+      if (tab->type == JT_ALL && ...)
+          DBUG_RETURN(0);
    }
    DBUG_RETURN(1);
}
int create_sort_index()
{
    if ((order != join->group_list || ... &&
        test_if_skip_sort_order(...))
      DBUG_RETURN(0);
    table->sort.found_records=filesort(thd,
        table,join->sortorder, ...);
}
```

The new control flow can change the function return value, which later affects whether an expensive path (with *firesort* call) will be taken or not

MySQL	PostgreSQL	Chrome
21 (42%)	9 (36%)	16 (64%)
8 (16%)	6 (24%)	4 (16%)
6 (12%)	5 (20%)	2 (8%)
15 (30%)	5 (20%)	3 (12%)

Performance sensitive condition

What the problematic change modifies?



```
uint make_join_readinfo(JOIN *join, ulonglong options)
{
    for (i=join->const_tables ; i < join->tables ; i++) {
        JOIN_TAB *tab=join->join_tab+i;
+       if (table->s->primary_key != MAX_KEY &&
+           table->file->primary_key_is_clustered())
+         tab->index= table->s->primary_key;
+       else
          tab->index=find_shortest_key(table, ...);
    }
}
int join_read_first(JOIN_TAB *tab)
{
    if (!table->file->inited)
       table->file->ha_index_init(tab->index, tab->sorted);
}
```

The new logic prefers clustered primary index over secondary ones. It degrades performance for certain workloads.

Performance critical variable

MySQL	PostgreSQL	Chrome
[illegible]	[illegible]	[illegible]
8 (16%)	6 (24%)	4 (16%)
6 (12%)	5 (20%)	2 (8%)
15 (30%)	5 (20%)	3 (12%)

How a change impacts performance?

Type of performance impact		MySQL	PostgreSQL	Chrome
Direct		34 (68%)	11 (44%)	12 (48%)
Indirect	Via function return value	7 (14%)	7 (28%)	3 (12%)
	Via function parameter	5 (10%)	4 (16%)	1 (4%)
	Via class member	1 (2%)	1 (4%)	3 (12%)
	Via global variable	1 (2%)	0 (0%)	1 (4%)
	Others	2 (4%)	2 (8%)	5 (20%)

Outline

- Understanding real world performance regression issues
- Performance risk analysis design
- Implementation: PerfScope
- Evaluation
- Conclusion

Performance Risk Analysis (PRA)

- Goal: statically analyze code change's risk in incurring performance regression
- Two pieces of information:
  - ☐ Cost of changed operation
  - ☐ Execution frequency of changed operation

Static cost model

```
class CostModel {
    protected:
        virtual unsigned getArithmeticInstrCost(…);
        virtual unsigned getMemoryOpCost(…);
        virtual unsigned getCallCost(…);
         …
    public:
        virtual unsigned getInstructionCost(…);
        virtual unsigned getBasicBlockCost(…);
        virtual unsigned getLoopCost(…);
        virtual unsigned getFunctionCost(…);
}
```

Static cost model

```
class CostModel {
    protected:
        virtual unsigned getArithmeticInstrCost(…);
        virtual unsigned getMemoryOpCost(…);
        virtual unsigned getCallCost(…);
         …
    public:
        virtual unsigned getInstructionCost(…);
        virtual unsigned getBasicBlockCost(…);
        virtual unsigned getLoopCost(…);
        virtual unsigned getFunctionCost(…);
}
```

Execution frequency estimation

- Static loop iteration count estimation
    - If cannot determine -> frequent
- Recursive function -> frequent
- Inter-procedural

Risk matrix

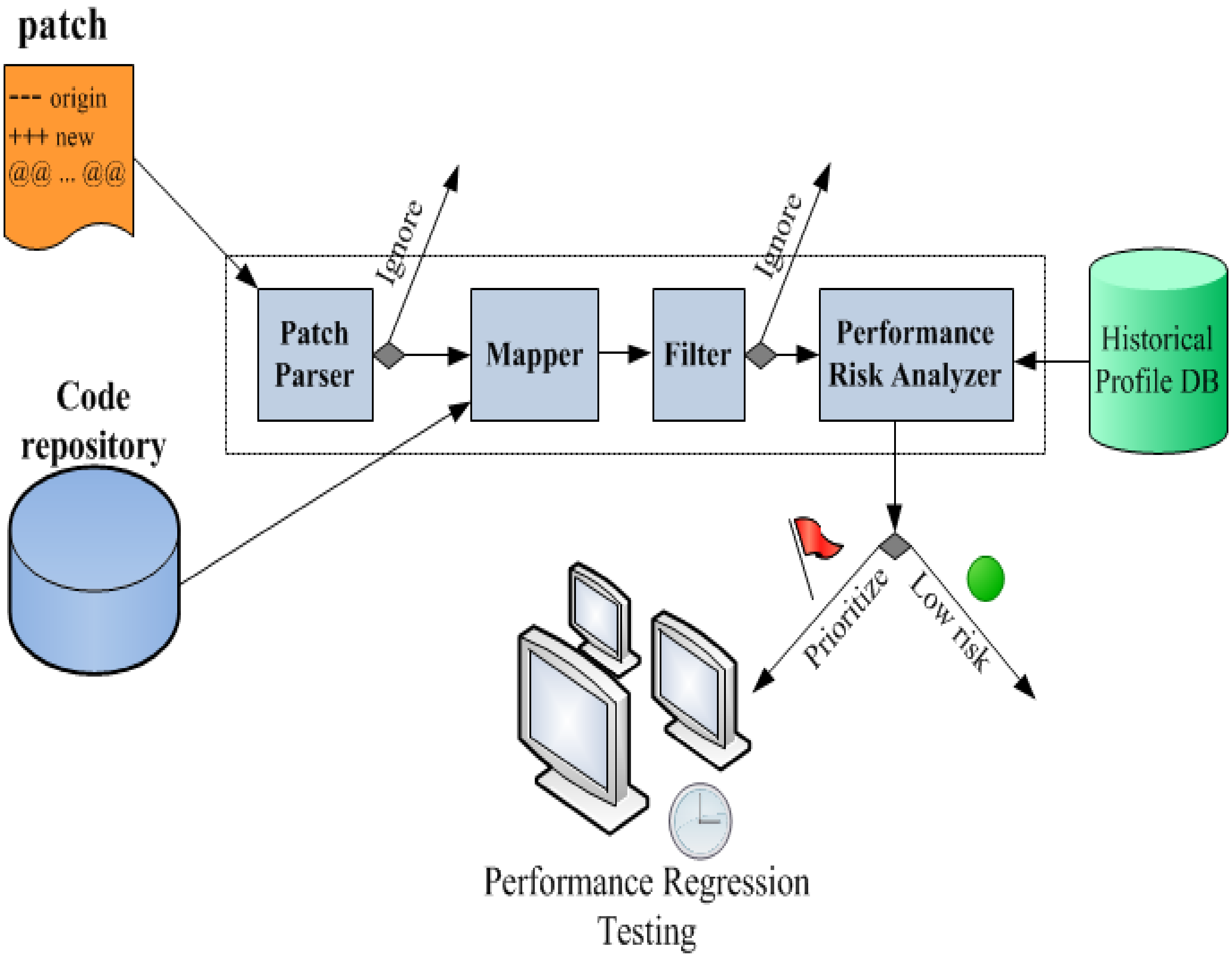
Cost	**Frequency**		
	Frequent	**Normal**	**Rare**
Expensive	Extreme	High	Moderate
Normal	High	Moderate	Low
Minor	Moderate	Low	Low

Performance risk matrix given cost and frequency information

Outline

- Understanding real world performance regression issues
- Performance risk analysis design
- Implementation: PerfScope
- Evaluation
- Conclusion

PerfScope architecture



PerfScope

- On top of LLVM infrastructure
- Currently support C/C++
- Open sourced in http://cseweb.ucsd.edu/~peh003/perfscope

Outline

- Understanding real world performance regression issues
- Performance risk analysis design
- Implementation: PerfScope
- Evaluation
- Conclusion

Evaluation on studied perf. regression commits

Software	Problematic Commits	Recommended
MySQL	39	35
PostgreSQL	25	23
Total	**64**	**58 (91%)**

Evaluation on new perf. regression commits

- 600 new commits from 6 popular, large-scale software
- Obtained "ground truth" by running standard perf. testing suite

Software	LOC	Studied?
MySQL	1.2M	Yes
PostgreSQL	651K	Yes
GCC	4.6M	**No**
V8	680K	**No**
Squid	751K	**No**
Apache	220K	**No**

Evaluation on new perf. regression commits

Software	Test Commits	Risky Commits	Recommended (Reduction)	Miss (Coverage)
MySQL	100	9	22 (78%)	1
PostgreSQL	100	6	16 (84%)	0
Squid	100	5	12 (88%)	0
Apache	100	6	14 (86%)	0
Total	**600**	**39**	**100 (83%)**	**2 (95%)**

PerfScope can reduce at least 78% of the performance regression testing candidates and is still able to alarm 95% of the risky ones.

Running time of PerfScope

Software	LOC	Analysis Time (Seconds)
MySQL	1.2M	235
PostgreSQL	651K	194
GCC	4.6M	289
V8	680K	344
Squid	751K	34
Apache	220K	9

Outline

- Understanding real world performance regression issues
- Performance risk analysis design
- Implementation: PerfScope
- Evaluation
- Conclusion

Limitations and future work

- Cost modeling is simple
- No offsetting for delete/replace changes
- Mainly for CPU cost
  - Can be extended for I/O
- Combine with perf. test case prioritization
  - Already know which code region is risky, associate with coverage information.

Conclusion

- Software evolves fast that can inevitably worsen perf..
- Performance testing is an effective way to catch performance regression but it is costly.
- We propose performance risk analysis to prioritize performance testing target.
- Evaluation shows our tool is light-weight and effective in recommending performance-risky commits
- http://cseweb.ucsd.edu/~peh003/perfscope

Thanks!

The authors are unable to attend the conference and do Q&A due to Visa issues ☹
If you have any questions, please reach Peng at
ryanhuang@cs.ucsd.edu