# DATA ANALYTICS USING DEEP LEARNING

## GT 8803 // FALL 2018 // VARSHA ACHAR

LECTURE #07: UNDERSTANDING DATABASE PERFORMANCE INEFFICIENCIES IN REAL-WORLD APPLICATIONS

CREATING THE NEXT®

# TODAY'S PAPER

- Understanding Database Inefficiencies in Real-world Applications

- Authors:
  - Cong Yan and Alvin Cheung from University of Washington
  - Junwen Yang and Shan Lu from University of Chicago

- CIKM 2017: International Conference on Information and Knowledge Management

# TODAY'S AGENDA

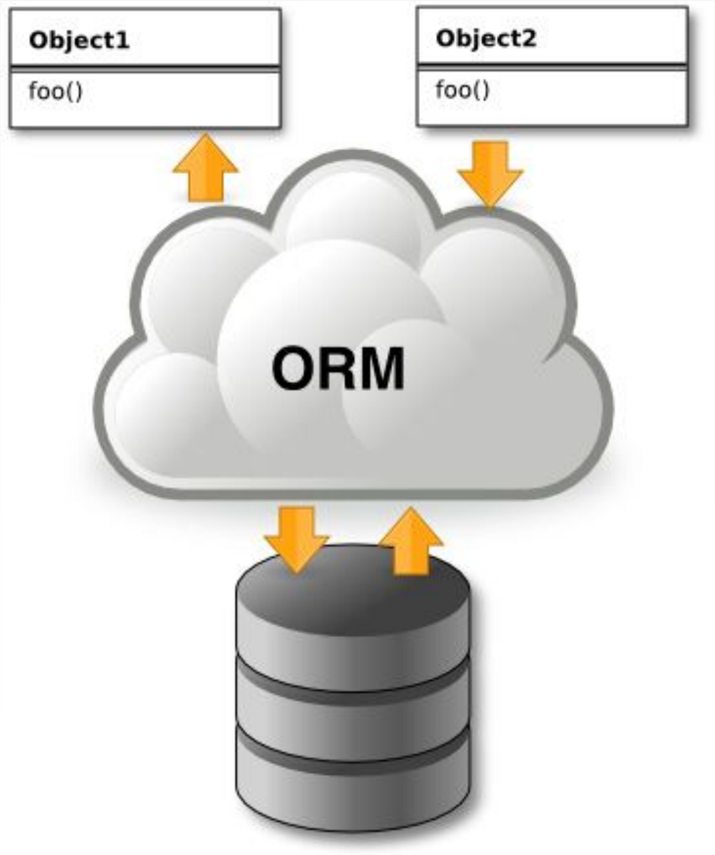
- Problem Overview
- Related Concepts
- Key Idea
- Technical Details
- Proposed Optimizations
- Discussion

# PROBLEM OVERVIEW

- Database-backed web applications today are built on ORM (Object Relational Mapping) frameworks.
- This eases development, but comes at a performance cost.
- This paper aims at identifying inefficiencies in such applications and suggest ways to increase performance.

# RELATED CONCEPTS: ORM

- **ORM**: Object relational mapping is a programming technique for converting data between relational and object-oriented data models.




API calls → Translation by ORM in DBMS queries → Result as objects → Application

Image from here

# RELATED CONCEPTS: MVC

- **MVC**: Model-view-controller architecture divides the application into three interconnected parts.
  - Model: manages data
  - View: Output representation
  - Controller: Intermediate that takes user input and passes it to the model.
- An advantage is code reusability.

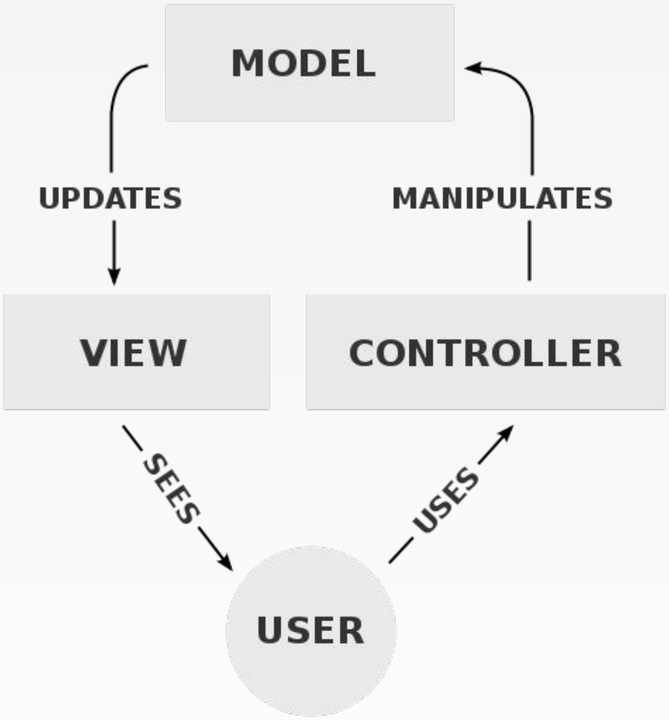



Image from here

# RELATED CONCEPTS: STATIC ANALYSIS AND AFGs

- **Static program analysis** refers to analyzing computer programs without actually executing the program.
- In this paper, their static program analyzer generates **Action Flow Graphs, or AFGs.** These are flowcharts that contain control-flow and data-flow for each action. It also contains ORM specific information inside and across different actions.

# KEY IDEA

- Common performance inefficiencies:
  - Poor database design
  - Coding patterns that lead to the ORM generating inefficient queries
  - Redundant computation as a result of lack of caching results
- Examined real world applications
  - Detected inefficiencies by generating AGFs using static program analysis
  - Proposed and manually applied optimizations to applications
  - This increased overall performance

# TECHNICAL DETAILS

- Chose 27 real world open-source applications from a wide range of domains.
  - Criteria: popularity on GitHub, no. of commits, no. of contributors, and application category.

- Ruby on Rails

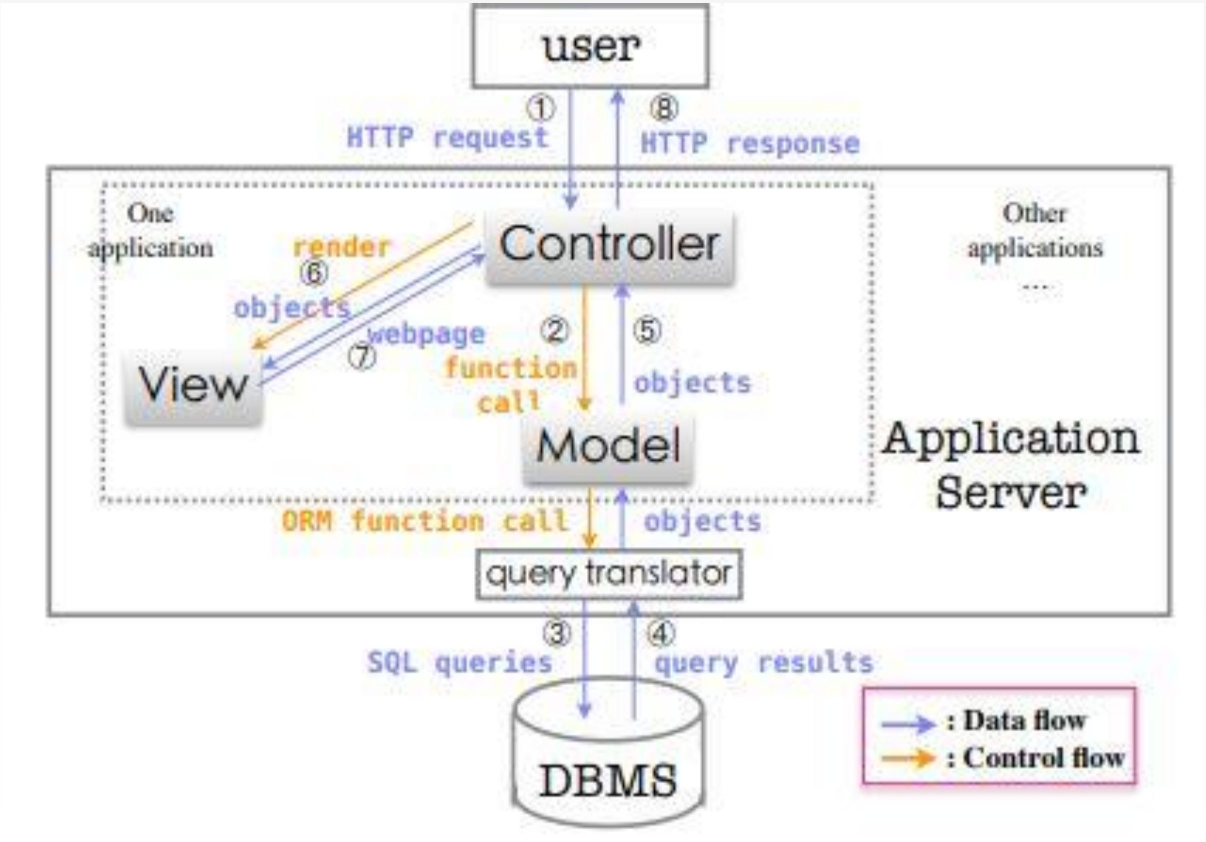



Image from here

# TECHNICAL DETAILS

- Classes in the ‘Model’ map to tables in the DBMS.
- Relationships in model classes are similar to the relationship between tables. (*has_many, belongs_to*)

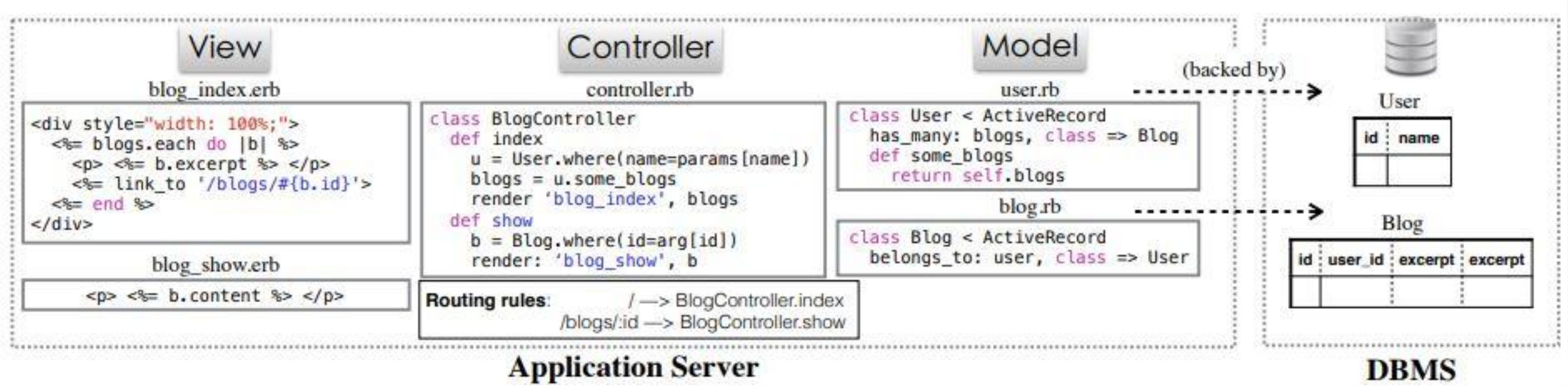



Image from here

# TECHNICAL DETAILS

- Performed Static Analysis to generate AFGs.
- Next action edge is determined based on possible user interactions - submitting a form or clicking a URL.
- In addition, 7 out of 27 applications were profiled with "synthetic data" to evaluate the optimizations.

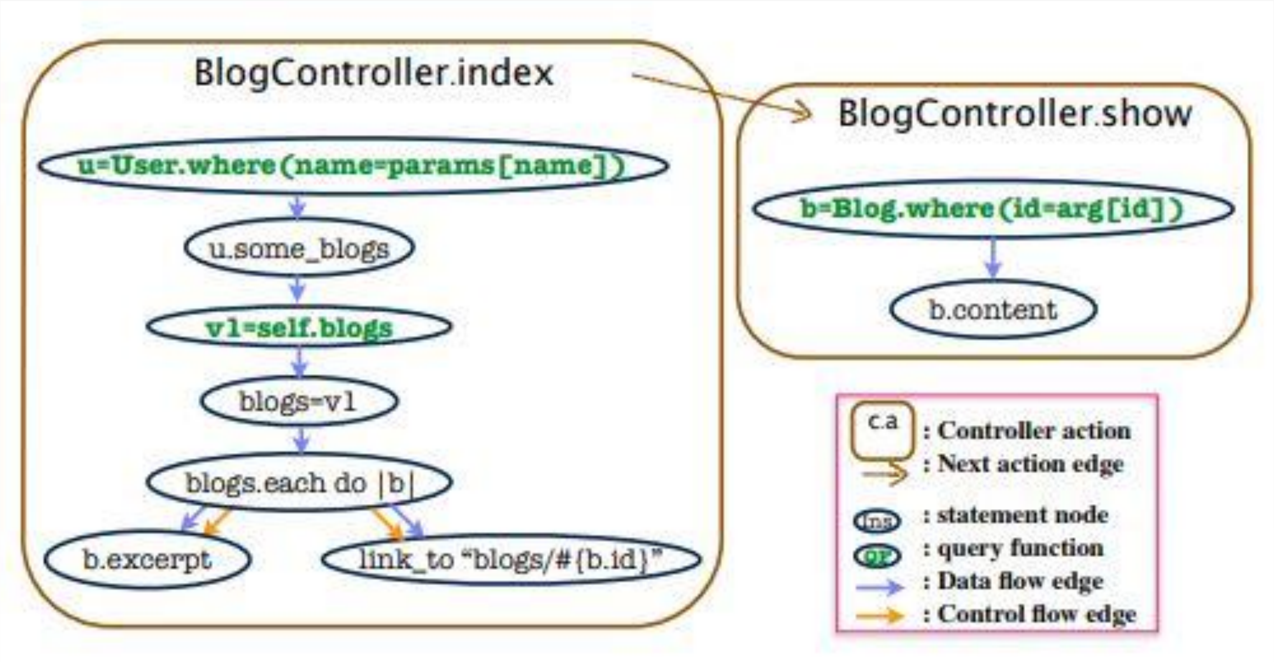



Image from here: Action Flow Graph (AFG)

# SINGLE ACTION ISSUES

★ Performance issues within a **single** action:

- Query translations
  - Caching common subexpressions
  - Fusing queries
  - Eliminating redundant data retrieval
- Rendering query results

# PROPOSED OPTIMIZATIONS

<u>CACHING COMMON SUBEXPRESSIONS:</u>

- It was found that queries shared common subexpressions.
- Caching these results reduced execution time by 67%
- An example of two queries sharing a common subexpression:

**Query 1**: SELECT name FROM employees WHERE state = “GEORGIA” AND salary <> 60000 ORDER BY emp_id ASC

**Query 2**: SELECT name FROM employees WHERE state = “GEORGIA” AND age = 50 ORDER BY emp_id ASC

# PROPOSED OPTIMIZATIONS

FUSING QUERIES:

- A lot of queries were evaluated to be used in subsequent queries.
- To understand how query results are used, dataflow is traced from each query node in the AFG until a query function node is reached, or the node has no outgoing dataflow edge.
- Examining “redmine”: 33% queries are only used for subsequent queries.
- Less transfer of data between DBMS and application.
- Issues? Repeated execution and optimizer.

# PROPOSED OPTIMIZATIONS

## REDUNDANT DATA RETRIEVAL:

- Default: SELECT *, unless explicitly mentioned.
- Many fields are not used in subsequent computation.
- Around 63% is not used.

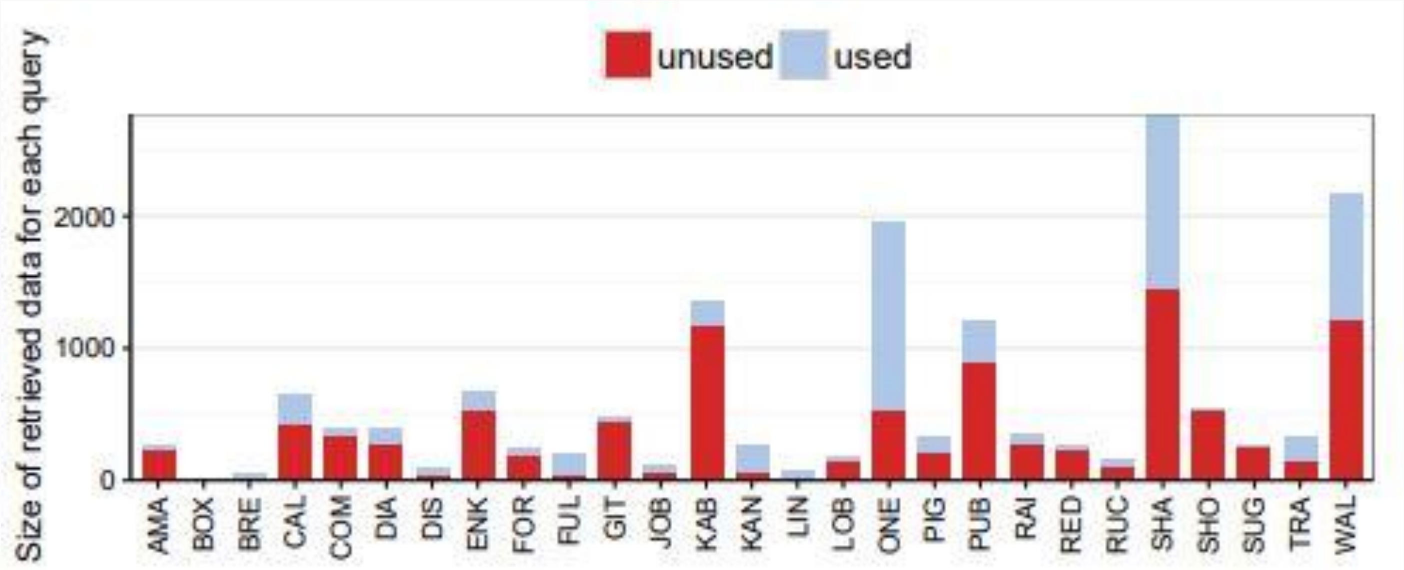



**Image from here: Used and unused retrieved data across the 27 applications.**

# PROPOSED OPTIMIZATIONS

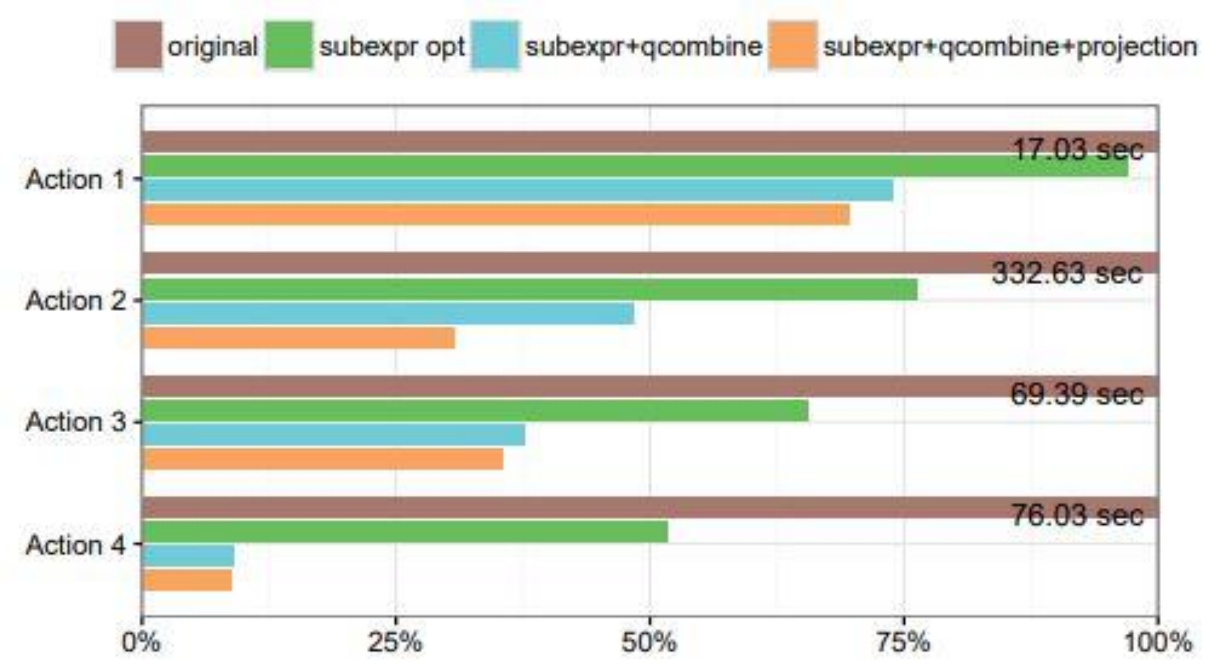
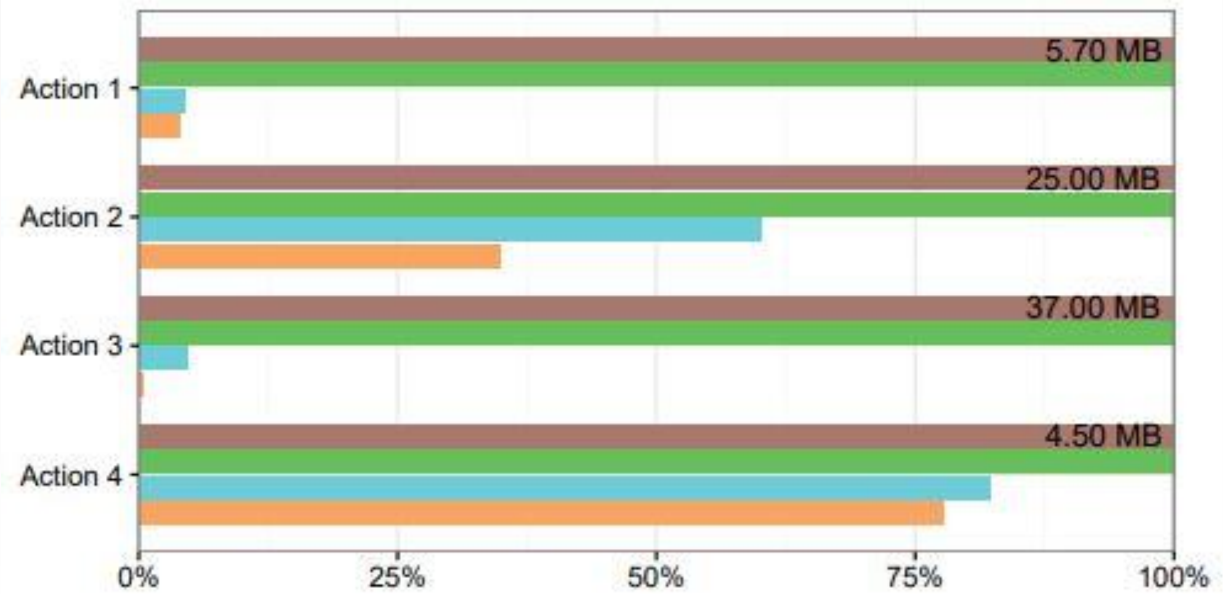



Image from here: Performance gain after combining optimizations.
Reduction of query time up to 91%

Image from here: Transfer size reduction.
More than 60% reduction of transfer data in Actions 1, 2, and 3.



# PROPOSED OPTIMIZATIONS

## RENDERING QUERY RESULTS:

- Problem - Loops, loops, loops!
- Larger the DB, longer it takes to render results.
- Bounded results: LIMIT, single value (COUNT), single record.
- Evaluation shows that 36% queries return unbounded results.
- Solution: Pagination and incremental loading.
- Rendering time reduction by around 85%.

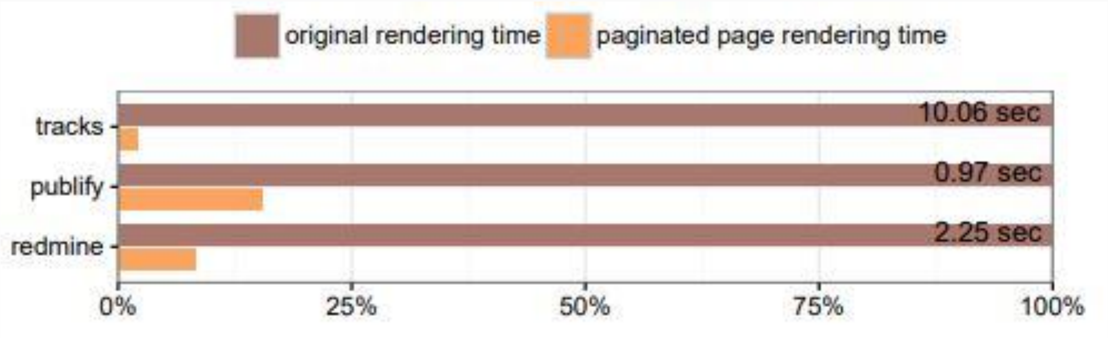



**Image from here: Evaluation after pagination.**

# MULTIPLE ACTION ISSUES

★ Performance issues within a **multiple** actions:

- Caching
- Storing data on the disk
  - Partial evaluation of selections
  - Partial evaluation of projections
  - Table denormalization

# PROPOSED OPTIMIZATIONS

CACHING: *(previous-current action pair)*

- Same queries across actions - checking user permission, partial page layout.
- Focus on syntactically equivalent queries (20%) and queries that share the same template (31%).

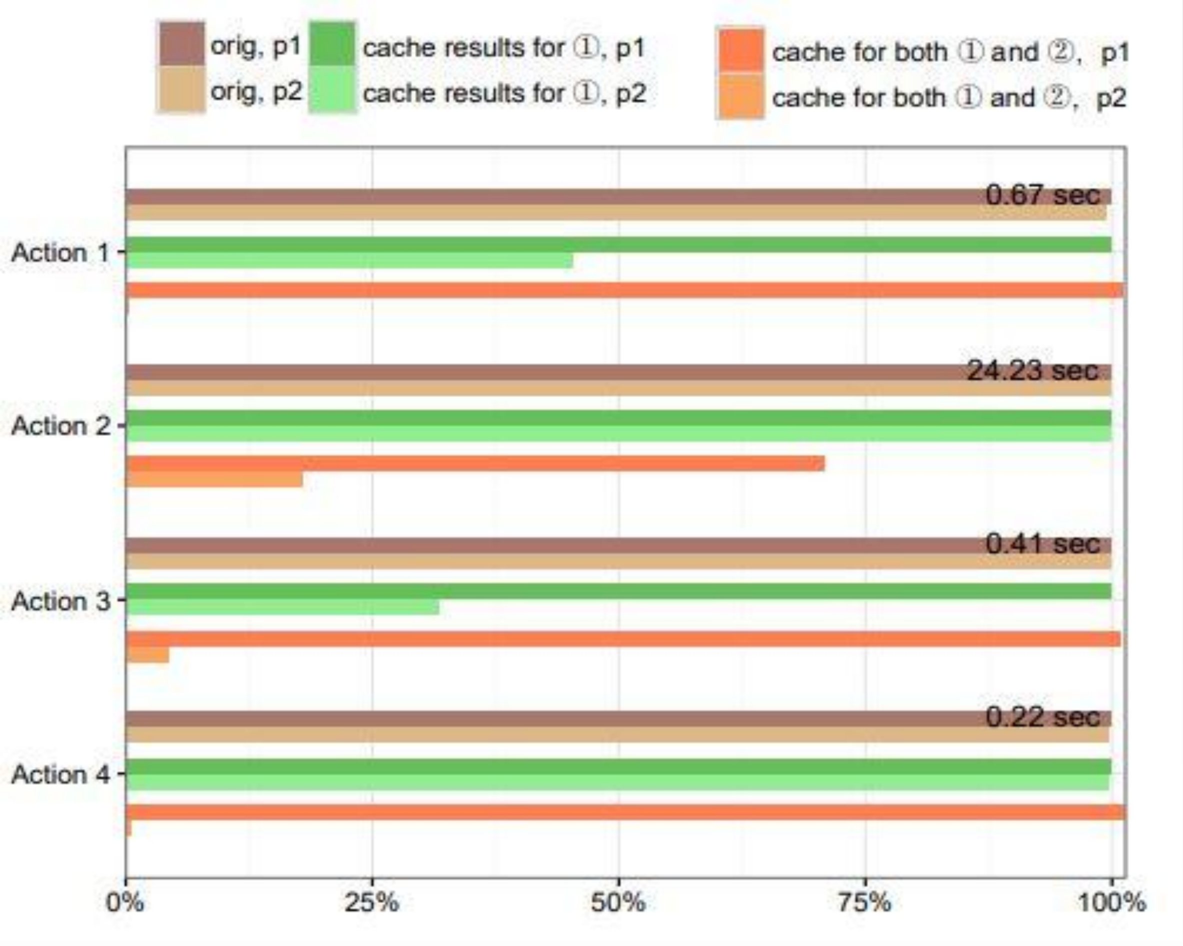



**Image from here: Caching evaluation with pages p1, p2. Baseline is orig p1.**

# PROPOSED OPTIMIZATIONS

## PARTIAL EVALUATION OF SELECTIONS:

- Programmatically generated queries usually have constant values as parameters. (33%)
- Key idea: Partially evaluate query with known values and store. Remaining user input dependent portion of the query is evaluated during runtime.
- Consider: Query Q on Table T and a constant predicate p
  Partially evaluate Q by partitioning T row-wise into two tables - one satisfying p, and the other not. Rewrite Q to execute on partitioned table.
- For N queries with different p on one T, partition recursively ($2^N$ partitions)
- Static analysis shows an average split of 3.2 for each table.

# PROPOSED OPTIMIZATIONS

PARTIAL EVALUATION OF PROJECTIONS:

- Many queries only use a subset of all fields in a table. (61%)
- ORM frameworks map each class to a table by default - full row is retrieved.
- Larger fields are used by fewer queries compared to smaller fields.
- Co-locate fields used together in order to partially evaluate projections.
- Vertically partition and rewrite queries.
- What if a query used all fields? Join the tables - added overhead.
- **But**, this could be trivial if the key for join is indexed.

# PROPOSED OPTIMIZATIONS

TABLE DENORMALIZATION:

- Essentially means that joins can also be partially evaluated.
- Stored pre-joined tables leads to performance gain, as joins are computationally expensive!
- After performing static analysis, it was found that 55% queries are joins and each join involves an average of 2.8 tables.
- Problems: duplicate data, slows down write queries and read queries.
- **But**, combining with vertical partitioning somewhat helps reduce data duplication.
- Only the fields used in the join query are denormalized to be stored in a table, others are kept in the original table.

# PROPOSED OPTIMIZATIONS

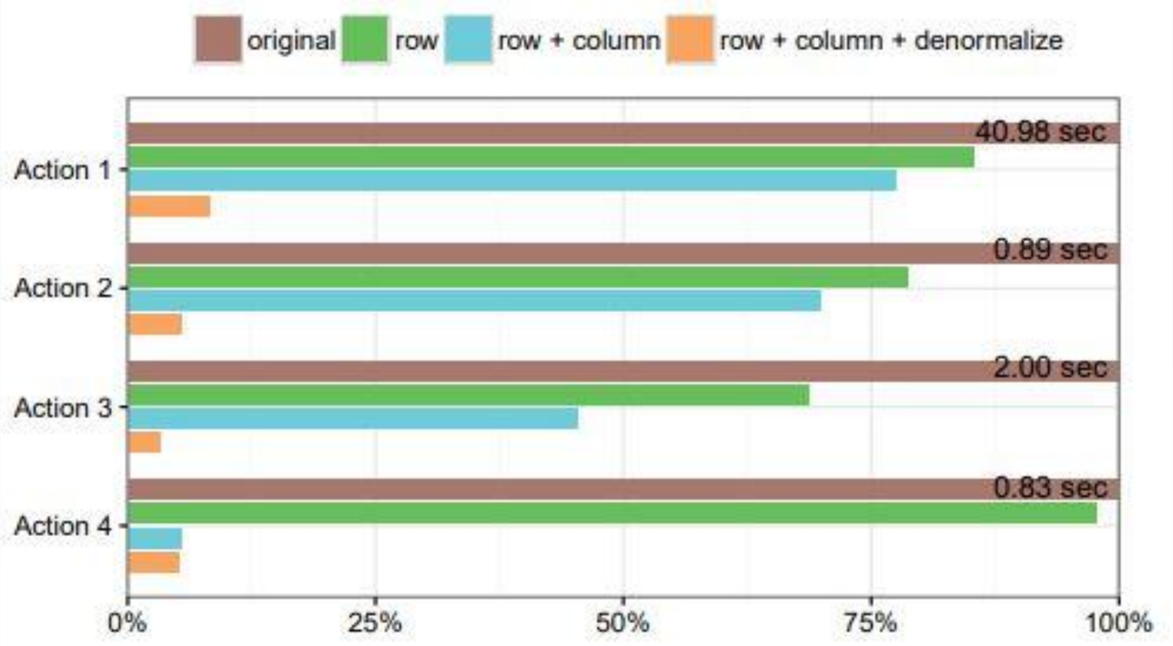
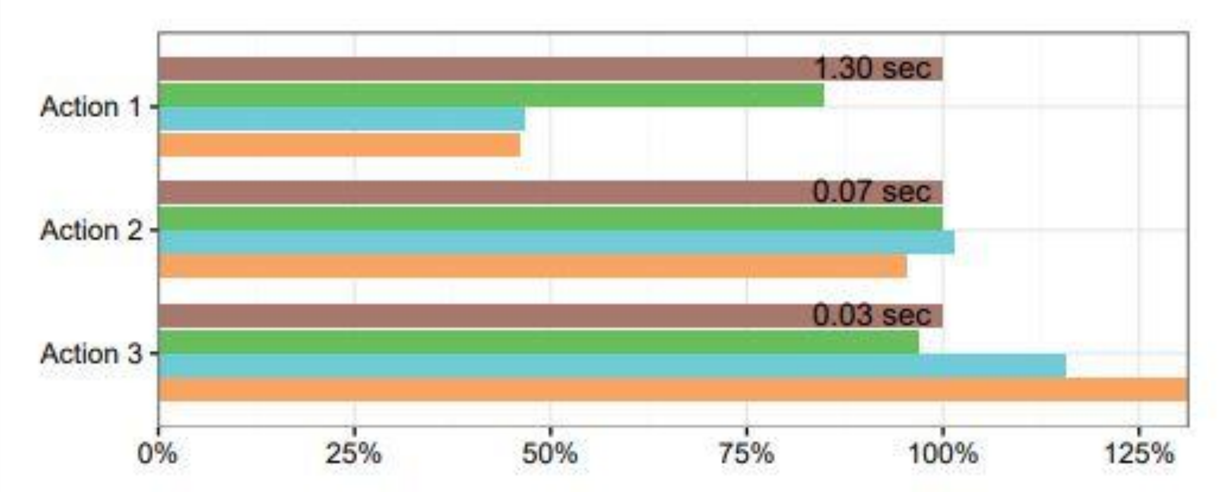



Image from here: Performance for GET actions, original and optimized.

Image from here: Performance for POST actions, original and optimized.



# PROPOSED OPTIMIZATIONS

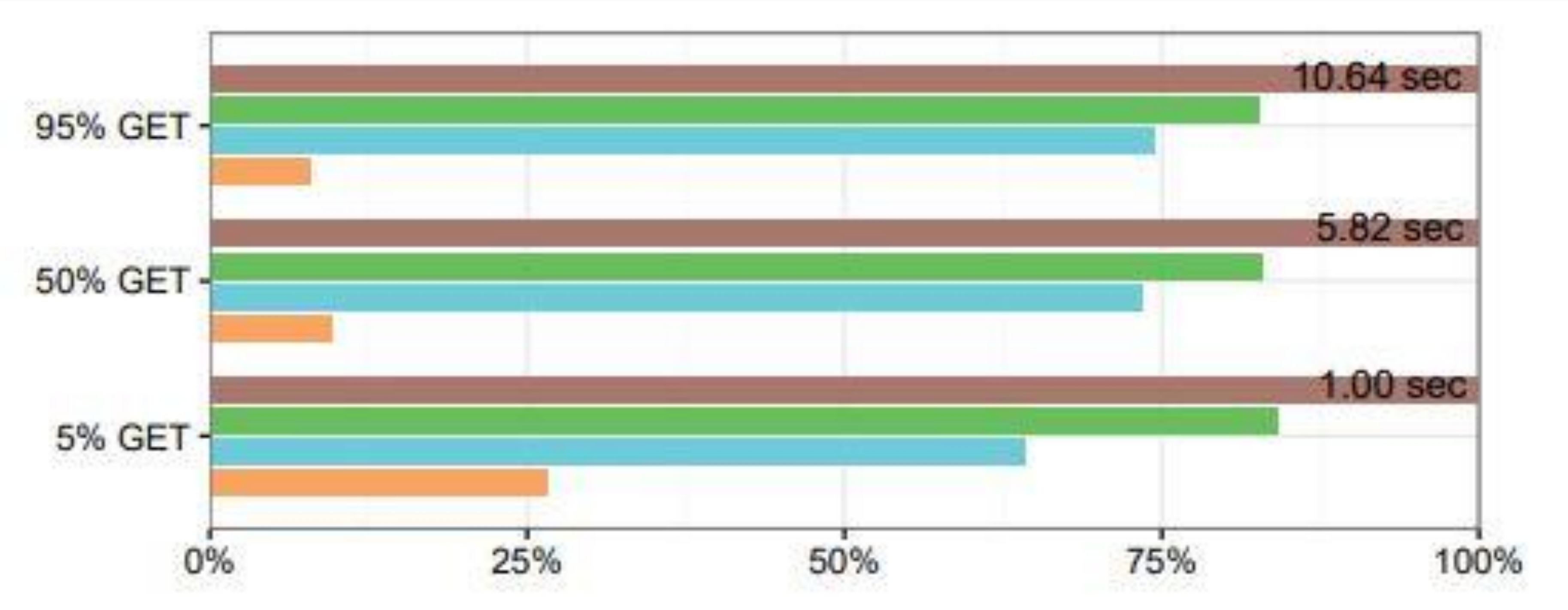



**Image from here: Performance for a mix of GET and POST actions, original and optimized.**

# DISCUSSION

- Strengths and weaknesses?
- Was it useful to know about these inefficiencies? Does it matter how the queries are executed?
- "Synthetic data"
- General enough? Will it work across all web frameworks?
- Will these techniques improve performance with changes in the type of DB? (MySQL vs DB2 vs Postgres)
- Any inspiration for future research?