Boston University

OPTIMIZING LLM QUERIES IN RELATIONAL DATA ANALYTICS WORKLOADS

CS561 - Student Presentation

Boston University

OVERVIEW

- Introduction & Background
- Motivation & Problem
- Solution
- Evaluation & Discussion

INTRODUCTION

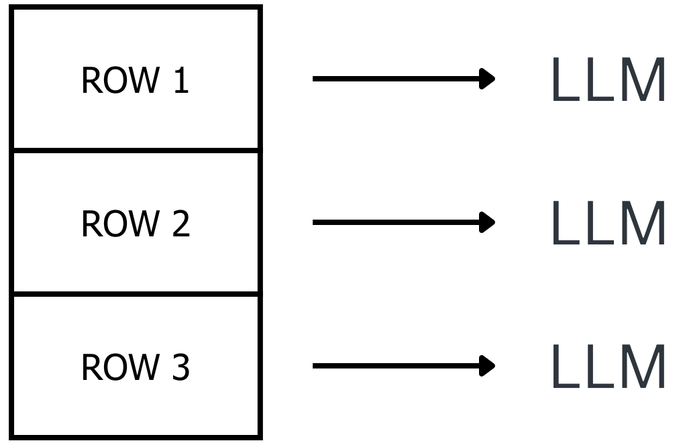
- LLMs are integrated into modern data systems (BigQuery, Databricks, Redshift)
- Enable tasks:
  → Classification
  → Summarization
  → Reasoning
- LLMs can be used directly inside SQL queries

WHAT IS AN LLM QUERY?

- LLM is invoked once per row
- Each row → independent LLM request

Example:

SELECT LLM(prompt, fields) FROM table



MILLIONS OF ROWS → MILLIONS OF LLM CALLS

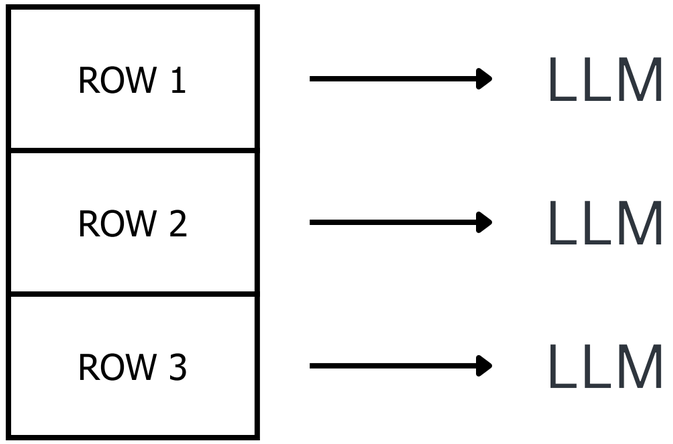
WHAT IS AN LLM QUERY?

```
SELECT user_id, request,
↪  support_response,
  LLM('Did {support_response} address
  ↪  {request}?', support_response,
  ↪  request) AS success
FROM customer_tickets
WHERE support_response <> NULL
```

Example:

request: "Battery drains fast"

support_response: "Try reducing apps"



MILLIONS OF ROWS → MILLIONS OF LLM CALLS

LLM INFERENCE PIPELINE

- **Prefill:** processes entire input prompt (expensive)
- **Decode:** generates output token-by-token

- Prefill dominates cost (~80–90%)
- Repeated prompts → repeated expensive computation

PROMPT → PREFILL → DECODE

↓

KV CACHE

LLM INFERENCE PIPELINE

PROMPT:

"SUMMARIZE PRODUCT IPHONE REVIEWS"

↓

READ (PREFILL)

[SUMMARIZE] → PROCESS

[PRODUCT] → PROCESS

[IPHONE] → PROCESS

[REVIEWS] → PROCESS

↓

GENERATE (DECODE)

"THESE REVIEWS SUGGEST..."

LLM INFERENCE PIPELINE

PROMPT:

"SUMMARIZE PRODUCT IPHONE REVIEWS"

↓

READ (PREFILL) → **Reading (prefill) the prompt is the most expensive part**

[SUMMARIZE] → PROCESS

[PRODUCT] → PROCESS

[IPHONE] → PROCESS

[REVIEWS] → PROCESS

↓

GENERATE (DECODE)

"THESE REVIEWS SUGGEST..."

LLM INFERENCE PIPELINE

PROMPT 1:
[SUMMARIZE] [PRODUCT] [IPHONE]
→ **STORED**

PROMPT 2:
[SUMMARIZE] [PRODUCT] [SAMSUNG]
→ **REUSE FIRST PART**

PROMPT → PREFILL → DECODE

↓

KV CACHE

STORE → **WHERE?** →

KV CACHE: REUSE PROMPT UNDERSTANDING INSTEAD OF RECOMPUTING IT

KEY OPTIMIZATION: KV CACHE

- Stores intermediate token computations
- Reuses shared prompt prefixes
- Avoids recomputation

Key Insight:

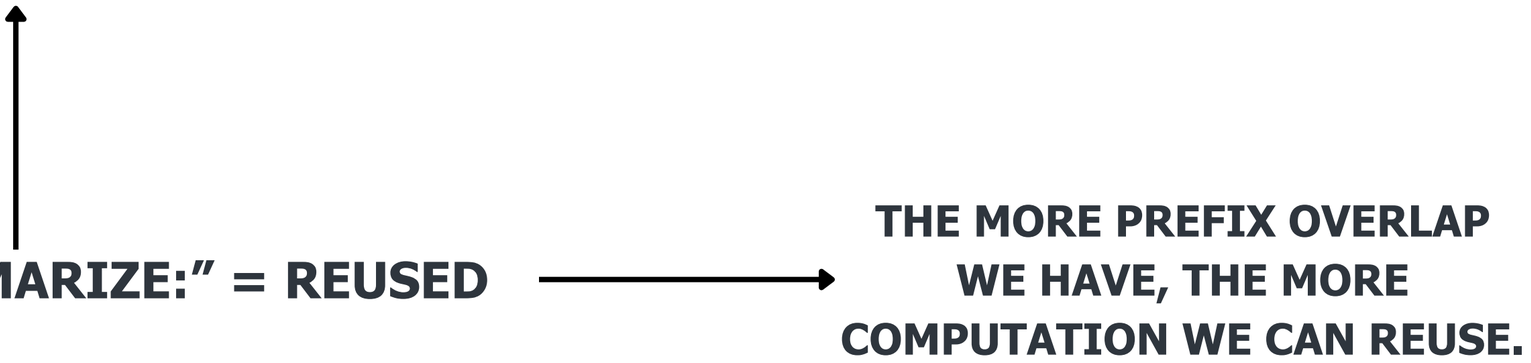
Performance depends on prefix similarity

KEY OPTIMIZATION: KV CACHE

EXAMPLE:

PROMPT 1: "SUMMARIZE: PRODUCT A..."

PROMPT 2: "SUMMARIZE: PRODUCT B..."



KEY OPTIMIZATION: KV CACHE

EXAMPLE PROMPT:
"SUMMARIZE PRODUCT IPHONE REVIEWS"

TOKENIZED:
["SUMMARIZE", "PRODUCT", "IPHONE", "REVIEWS"]

KEY OPTIMIZATION: KV CACHE

PROMPT 1:

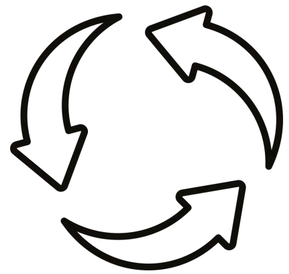
[SUMMARIZE] [PRODUCT] [IPHONE] [REVIEWS]

↓ ↓

KV CACHE STORES THESE

PROMPT 2:

[SUMMARIZE] [PRODUCT] [SAMSUNG] [REVIEWS]



REUSE:

- SUMMARIZE
- PRODUCT
- SAMSUNG (NEW) - NOT USED

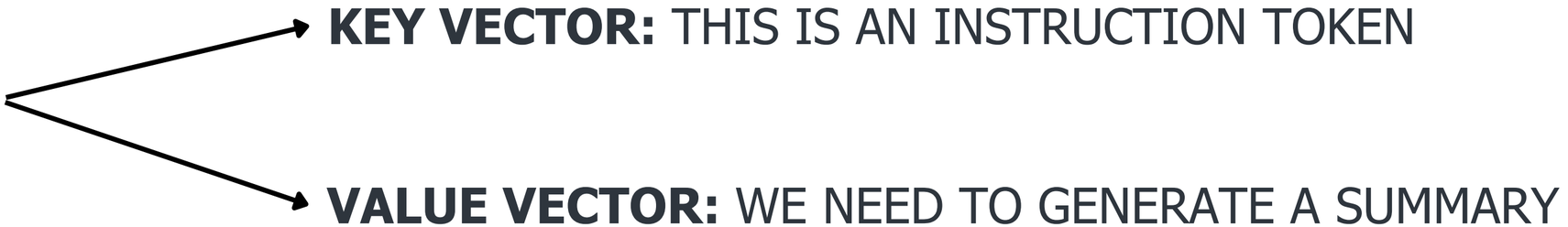
KEY OPTIMIZATION: KV CACHE

FOR EACH TOKEN, MODEL COMPUTES:

→ KEY VECTOR (WHAT TO ATTEND TO)

→ VALUE VECTOR (INFORMATION CONTENT)

THESE VECTORS ARE STORED IN KV CACHE



MOTIVATION: COST & LATENCY

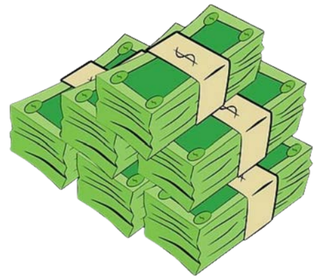
- LLM inference is slow and expensive

Example:

15GB data → ~1 day runtime → ~$10K cost

- Row-wise execution multiplies the cost
- Need system-level optimization

Millions of rows → Millions of LLM calls → $$$



PROBLEM IN CURRENT SYSTEMS

Current systems:

- Process rows independently
- Use fixed field ordering
- Do not group similar data

Result:

Low cache reuse

Redundant computation

RANDOM ROWS → LOW REUSE

GROUPED ROWS → HIGH REUSE

EVEN THOUGH DATA HAS SIMILARITIES, SYSTEMS FAIL TO EXPLOIT THEM.

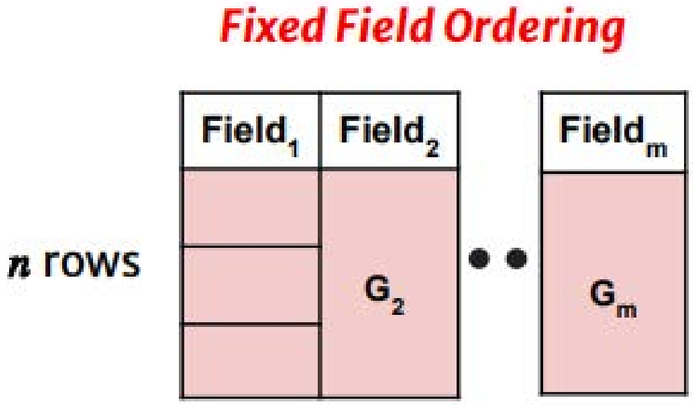
FIELD ORDERING IMPACT

Bad ordering: unique field first → no shared prefix across rows

Unique ID	Product	Category
123	iPhone	Electronics
456	iPhone	Electronics
789	iPhone	Electronics

Prefix reuse = 0

Green boxes = Cache Hits (GOOD)
Red boxes = Cache Misses (BAD)



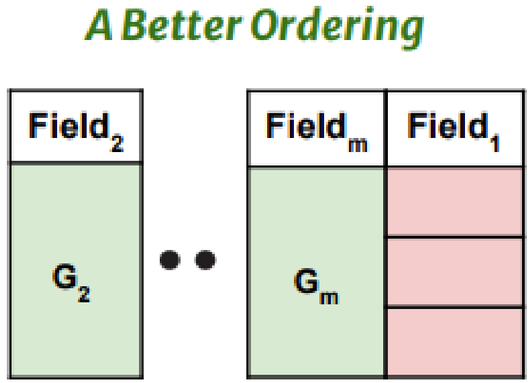
FIELD ORDERING IMPACT

Good ordering: Repeated field first → Strong prefix sharing

Product	Category	Unique ID
iPhone	Electronics	123
iPhone	Electronics	456
iPhone	Electronics	789

Prefix reuse = High

Green boxes = Cache Hits (GOOD)
Red boxes = Cache Misses (BAD)



PREFIX HIT COUNT (PHC)

- Measures prefix reuse between consecutive queries
- Higher PHC → more KV cache reuse

- **Goal: maximize PHC**

PHC = number of shared tokens between consecutive inputs

Example:
Query 1: "Summarize product iPhone..."
Query 2: "Summarize product Samsung..."

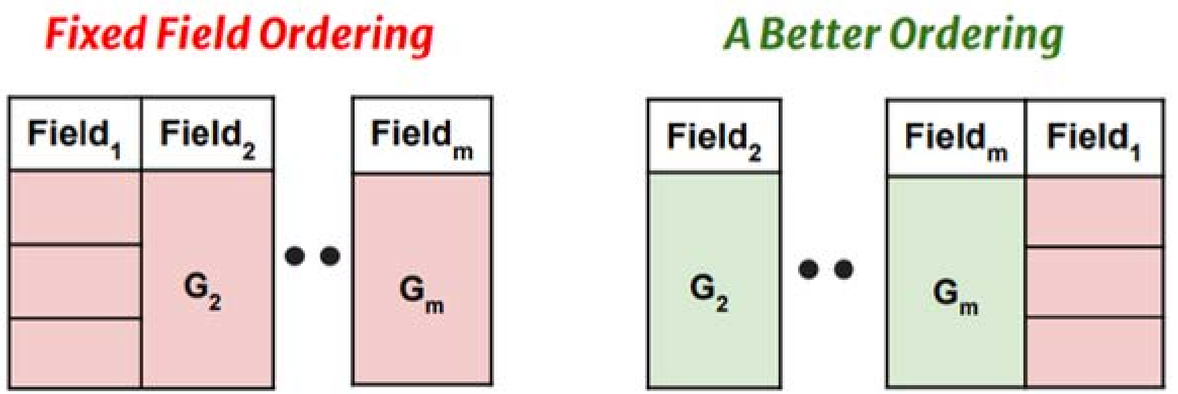
WHY FIXED ORDERING FAILS

- High-cardinality fields first → no prefix reuse
- Repeated fields appear later → wasted reuse
- Ignores data distribution

Result:

Low PHC

Poor performance



Same data, different ordering → huge performance difference

CACHE HIT IMPROVEMENT

Reordering dramatically increases prefix reuse

Method	Movies	Prods.
Original	35%	27%
GGR	86%	83%

Original(~30%) → GGR(~80%)

MORE CACHE HITS = LESS RECOMPUTATION = FASTER QUERIES

Method	Movies	Prods.	BIRD
Original	35%	27%	10%
GGR	86%	83%	85%

Key Insight:

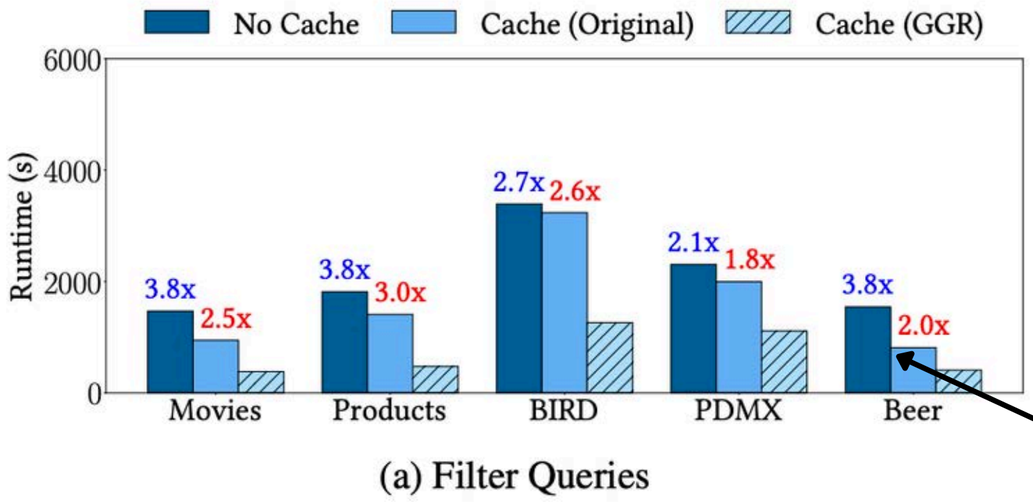
- Up to 75% higher cache hit rate
- Works across multiple datasets

Directly improves:

- Performance
- Cost

RUNTIME PERFORMANCE

Filter Queries: Reordering significantly reduces execution time



(a) Filter Queries

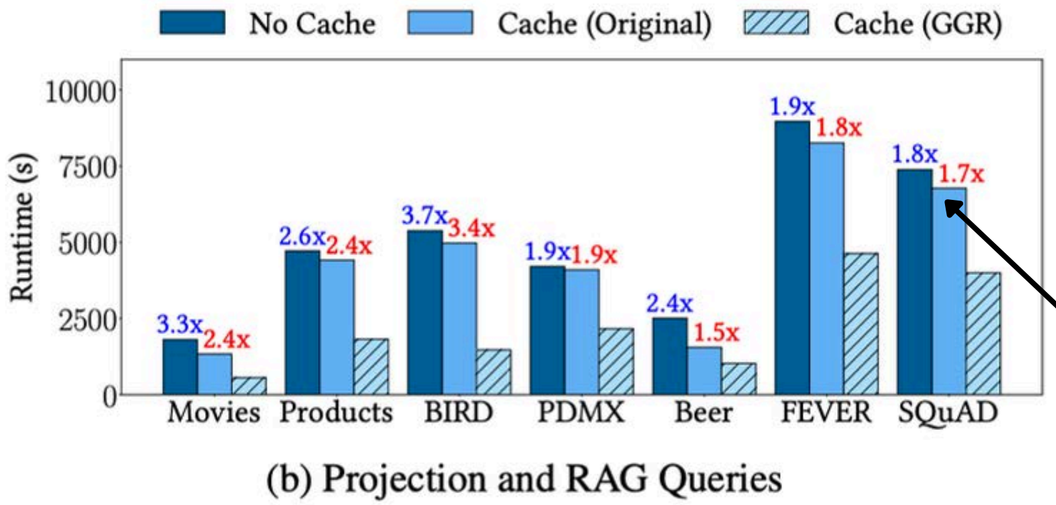
Key takeaways:

- No Cache = Slowest
- Cache (Original) = Moderate
- Cache (GGR) = Fastest

Up to 3.8x faster

RUNTIME PERFORMANCE

Projection and RAG Queries: Improvement persists across complex workloads



(b) Projection and RAG Queries

Key takeaways:

- Reordering improves performance even for semantic queries
- Benefits extend beyond simple filter queries

Up to 1.5x to 3.4x faster

RUNTIME PERFORMANCE

Projection and RAG Queries: Improvement persists across complex workloads

WHAT IS RAG (RETRIEVAL-AUGMENTED GENERATION)???

- Combines LLM + external data
- Retrieves relevant information before answering
- Improves accuracy and context

RUNTIME PERFORMANCE

Projection and RAG Queries: Improvement persists across complex workloads

User Query

↓

Retrieve relevant documents

↓

Add to prompt

↓

LLM generates answer

RUNTIME PERFORMANCE

Projection and RAG Queries: Improvement persists across complex workloads

WHAT DOES A RAG QUERY LOOK LIKE?

User Question:
"What do customers think about iPhone?"

Retrieved Data:
Review 1: "Battery is great"
Review 2: "Camera is amazing"

Final Prompt to LLM:

“Based on these reviews:
- Battery is great
- Camera is amazing”

RUNTIME PERFORMANCE

Projection and RAG Queries: Improvement persists across complex workloads

WHAT DOES A RAG QUERY LOOK LIKE?

User Question:
"What do customers think about iPhone?"

Retrieved Data:
Review 1: "Battery is great"
Review 2: "Camera is amazing"

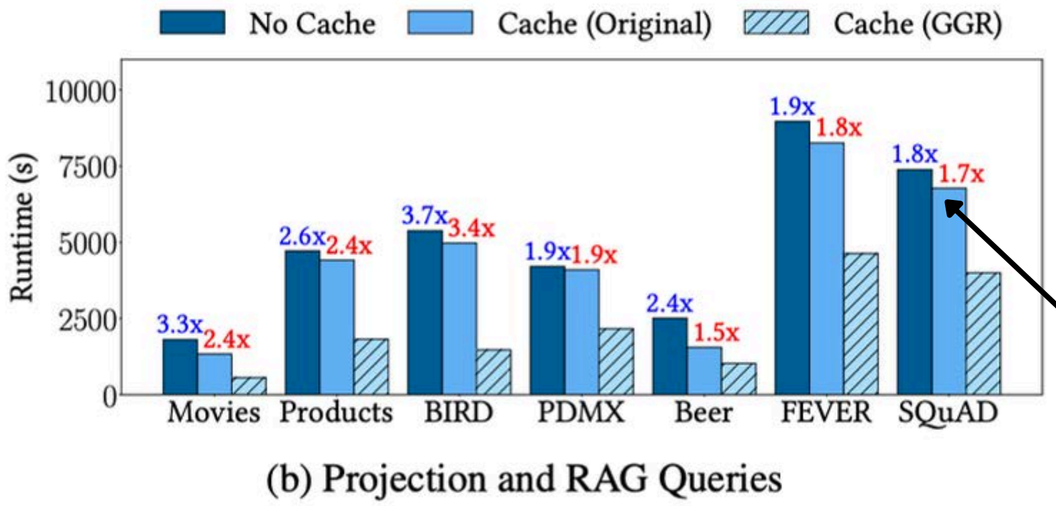
Final Prompt to LLM:

"Based on these reviews:
- Battery is great
- Camera is amazing"

Query → Retrieve → Augment → LLM

RUNTIME PERFORMANCE

Projection and RAG Queries: Improvement persists across complex workloads



(b) Projection and RAG Queries

- Improvements remain across complex workloads
- Gains slightly smaller but still significant
- Works for both projection and RAG queries

Key takeaways:

- Reordering improves performance even for semantic queries
- Benefits extend beyond simple filter queries

Up to 1.5x to 3.4x faster

CORE PROBLEM STATEMENT

Goal:

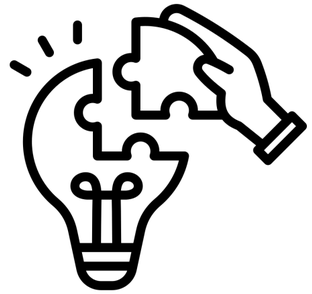
- Maximize reuse of LLM computation

How?

- Reorder rows to group similar data
- Reorder fields within rows to maximize prefix sharing

Key Idea:

- Optimize input structure instead of model architecture



Bad:

Random rows → Low reuse

Good:

Grouped rows → High reuse

SOLUTION?

Observation:

- Reordering significantly improves performance

Challenge:

- Finding the optimal ordering is computationally expensive

Solution:

- Efficient algorithms to approximate optimal ordering
  → OPHR
  → GGR

GOAL

Maximize PHC(Prefix Hit Count):

- “sum of squared prefix lengths across all rows where the LLM can reuse previously cached computation from a shared prompt prefix , rather than recomputing those tokens from scratch”

- **PHC = hit(row r)** = len(field_1)² + len(field_2)² + ... + len(field_c)²

- 10-token prefix → $10^2=100$ saved, not just 10

SOLUTION SCOPE

Row Reordering

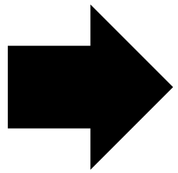
Product Type	Brand	Product Name	Product id
Mobile device	Apple	iPhone 15	1
Game console	Sony	Playstation 5	2
Mobile Device	Apple	iPhone 17	3
Game console	Sony	Playstation 4	4

Product Type	Brand	Product Name	Product id
Mobile device	Apple	iPhone 15	1
Mobile Device	Apple	iPhone 17	3
Game console	Sony	Playstation 5	2
Game console	Sony	Playstation 4	4

SOLUTION SCOPE

Field Reordering

Product ID	Product type	Product name	Price
1123	Apple	iPhone 16	$800
1234	Apple	iPhone 14	$500
1678	Apple	iPhone 15	$600
2340	Apple	iPhone 17	$1,000



Product type	Product name	Price	Product ID
Apple	iPhone 16	$800	1123
Apple	iPhone 14	$500	1234
Apple	iPhone 15	$600	1678
Apple	iPhone 17	$1,000	2340

What this means is that the **total number of orderings** for a table of n rows and m columns is (n! * (m)n) so we couldn't possibly brute force it we would need smart algorithms.

SOLUTION #1

OPTIMAL SOLUTION = OPHR (OPTIMAL PREFIX HIT RECURSION)

- Recursive divide and conquer algorithm that finds the optimal row and field arrangement that maximizes the PHC
- Try every possible (field, value) combination at each step and keep the split with the highest PHC

Split on (field = Category, value = Electronics)

Rat	Category	Brand	Review
5	Electronics	Apple	Bad..
4	Electronics	Samsung	Great...
3	Books	Penguin	Decent...
2	Books	Penguin	Great...

Table A: Category ≠ Electronics

Rat	Category	Brand	Review
3	Books	Penguin	Decent..
2	Books	Penguin	Long...

(recurse)

Table B: Category = Electronics (column removed)

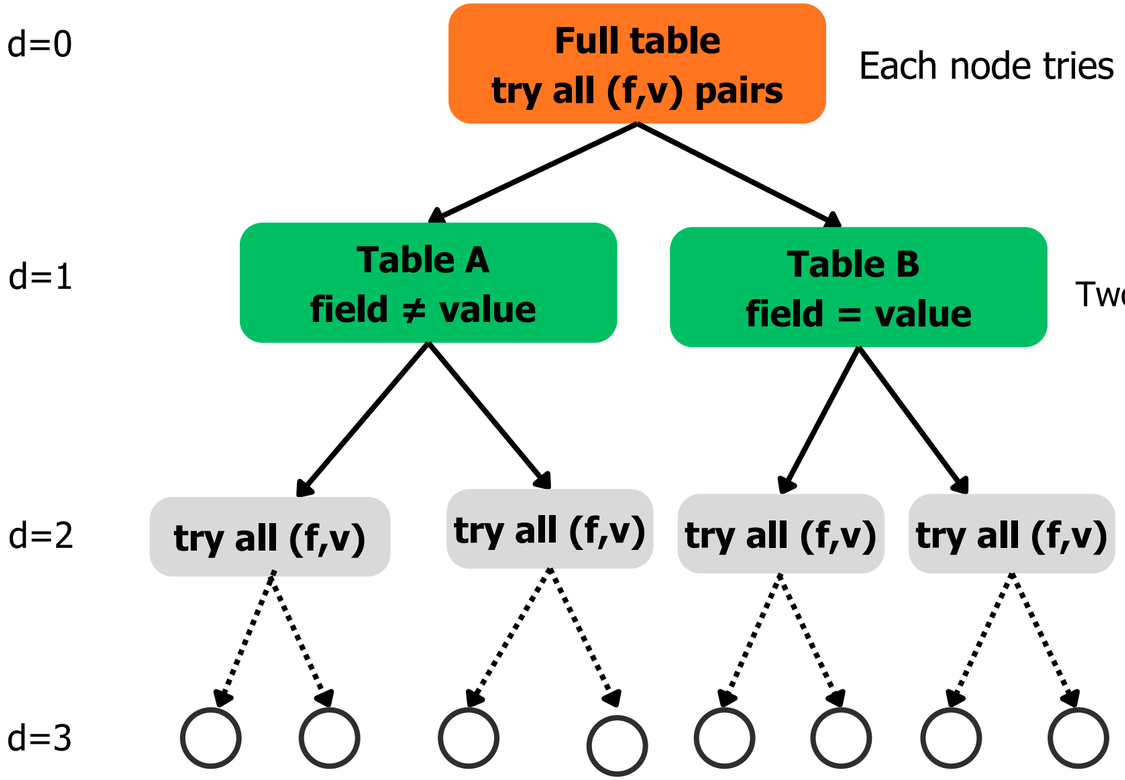
Rat	Brand	Review
5	Apple	Bad..
4	Samsung	Great...

(recurse)

PHC += |'Electronics'|²(shared prefix cached)

- Total PHC = PHC(Table A) + PHC(Table B) + contribution of v

HOWEVER, Optimal ≠ Feasible



- 2^d subproblems at depth d
- exponential total work
- impractical on real tables of large datasets

Boston University

SOLUTION #2

Practical Solution = GGR (Greedy Group Recursion)

- Same recursive structure as OPHR but instead of trying every possible (field, value) pair, it commits to the single best greedy choice at each step so **no backtracking**

HitCount per value

Field	Value	HitCount
Product type	Mobile Device	169
Product type	Game console	144
Brand	Apple	25
Brand	Sony	16
Product Name	iPhone	36

At each step, for every (field c, value v) pair compute:

- HitCount(v, c) = len(v)² × (count(v) - 1)
- Pick the pair (ĉ, v̂) with the highest score

Original table

Type	Brand	Name	Review
Mobile	Apple	iPhone 14	Great...
Mobile	Apple	iPhone 17	Faster...
Console	Sony	PS5	Fun...
Console	Sony	PS4	Old...

greedy pick: (Type, Mobile)

Table B — Type = Mobile (col removed)

Brand	Name	Review
Apple	iPhone 14	Great...
Apple	iPhone 17	Fast...

Table A — Type ≠ Mobile

Type	Brand	Name	Review
Console	Sony	PS5	Fun...
Console	Sony	PS4	Old....

(concatenate)

Final output (reordered)

Mobile . Apple	iPhone 14	Great...	← shared prefix
Mobile . Apple	iPhone 17	Fast...	← shared prefix
Console . Sony	PS5/PS4	Fun.../Old...	← Table A follows

Tradeoff: if two pairs have equal HitCount scores, greedy can pick suboptimally

GGR OPTIMIZATION

1. Functional Dependencies

Product id	Product name	Brand	Price
14657	iPhone 14	Apple	$500
(scan indepently)	(scan indepently)	(scan indepently)	(scan indepently)

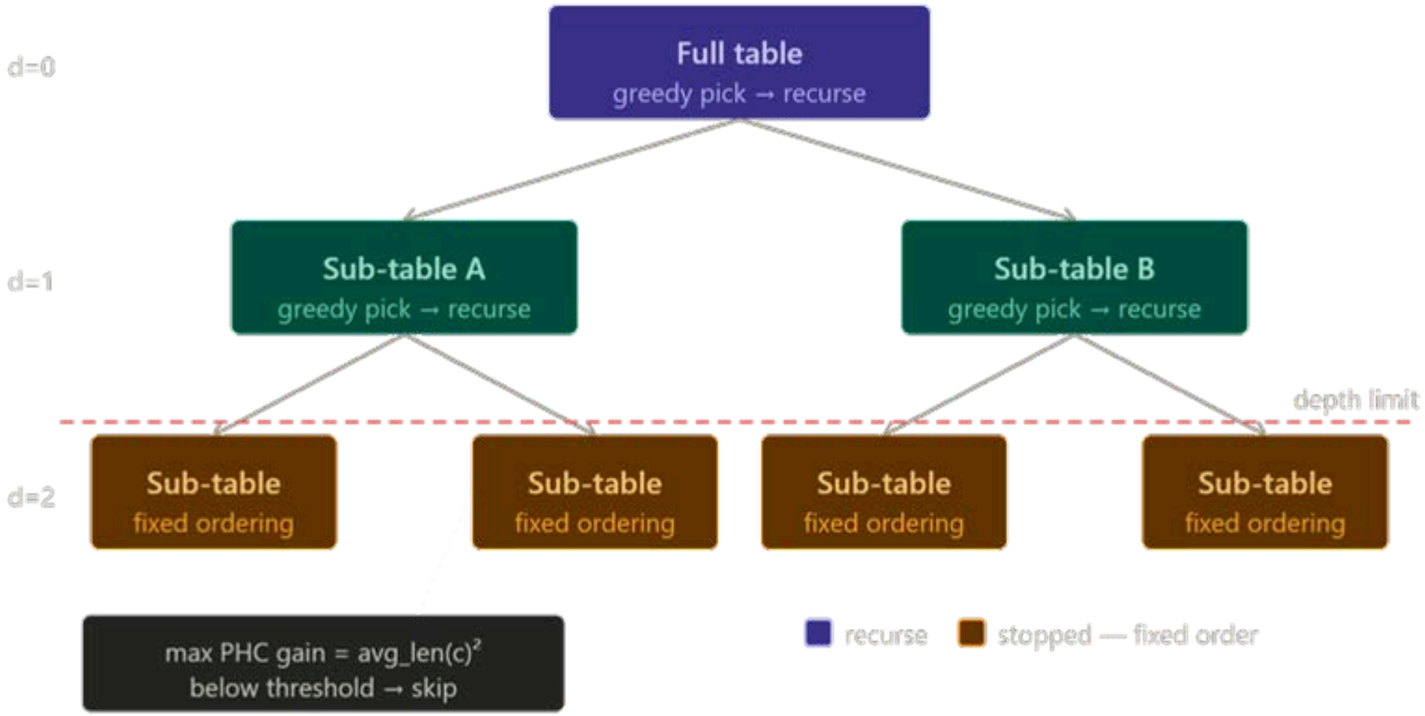
Notice Product_name always determines the Brand and Product_id

Product id	Product name	Brand	Price
14657	iPhone 14	Apple	$500
(scan as one)			(scan independently)

- This allows for **higher approximation quality** and **faster runtime**

GGR OPTIMIZATION

2. Table Statistics + Early Stopping



a. **Depth Limit -** stop after a fixed recursion depth, sub-tables at this point are small enough that a simple fixed ordering is nearly as good

b. **Hit count threshold -** estimate the maximum possible PHC gain from a sub-table using avg_len(c)².

Boston University

EVALUATION- DATASET

Rotten Tomatoes Movies (Classification)

Amazon Product Reviews (Sentiment/Extraction)

BIRD (Text-to-SQL)

Public Domain MusicXML (PDMX)

RateBeer Reviews

SQuAD (Question Answering)

FEVER (Fact Verification)

Boston University

EVALUATION - PROMPTS

(T1) LLM filter

WHERE clauses and use LLMs to categorize data

Given the following fields, answer in one word, 'Yes' or 'No', whether the movie would be suitable for kids. Answer with ONLY 'Yes' or 'No'

Boston University

EVALUATION

(T2) LLM projection

similar to a SQL SELECT statement to summarize data

Given information including movie descriptions and critic reviews, summarize the good qualities in this movie that led to a favorable rating.

Boston University

EVALUATION

(T3) Multi-LLM invocation

a filter followed by a projection, multi-step data processing

Given the following review, answer
whether the sentiment associated is 'POSITIVE' or
'NEGATIVE'. Answer in all caps with ONLY 'POSITIVE'
or 'NEGATIVE'

Boston University

EVALUATION

(T4) LLM aggregation

aggregate functions, like averaging sentiment scores given by LLMs

Given the following fields of a movie description and a user review, assign a sentiment score for the review out of 5. Answer with ONLY a single integer between 1 (bad) and 5 (good)

Boston University

EVALUATION

(T5) Retrieval-augmented generation (RAG)

external knowledge as context/ground truth
retrieves relevant document segments before generating answers

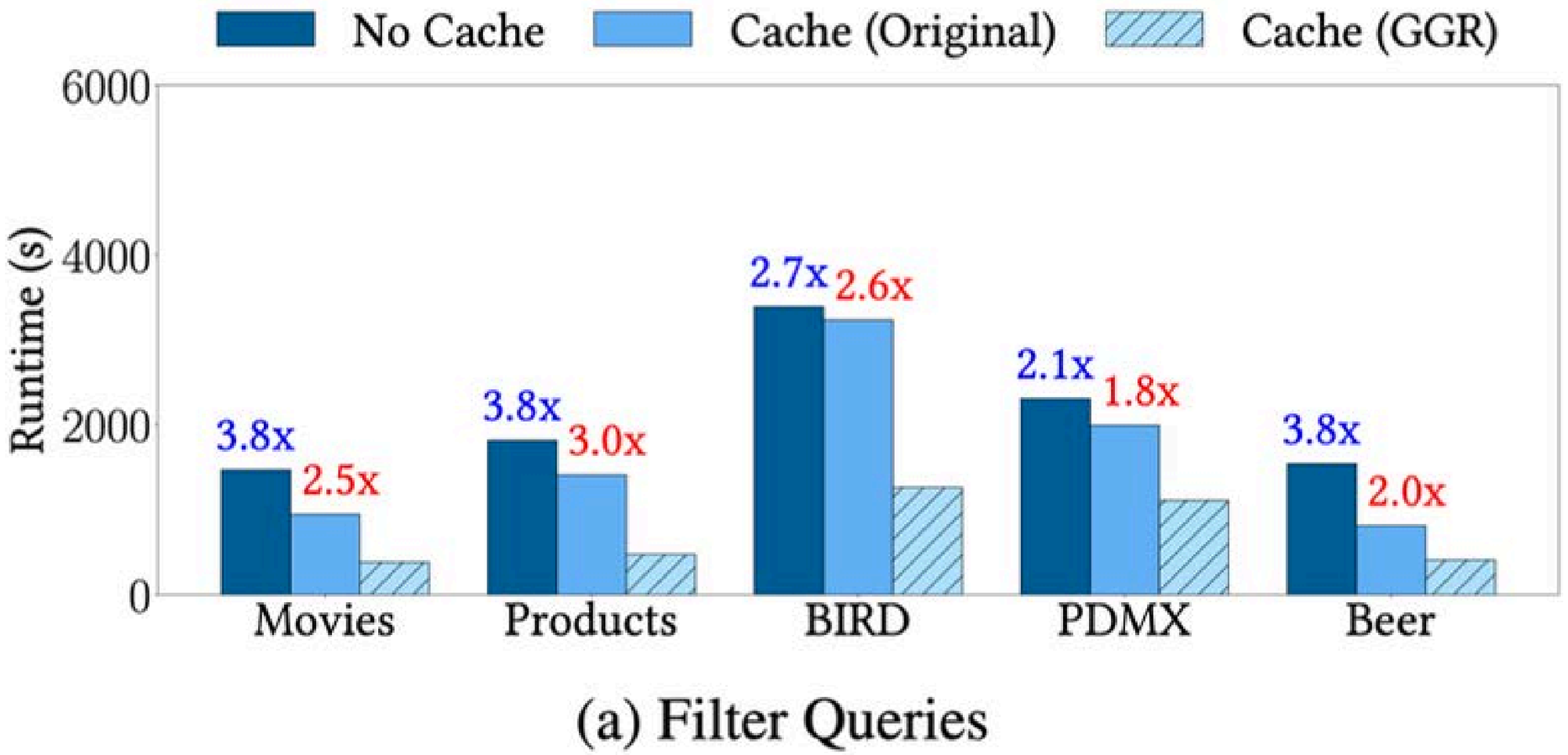
FEVER

You are given 4 pieces of evidence as {evidence1}, {evidence2}, {evidence3}, and {evidence4}. You are also given a claim as {claim}. Answer SUPPORTS if the pieces of evidence support the given {claim}, REFUTES if the evidence refutes the given {claim}, or NOT ENOUGH INFO if there is not enough information to answer. Your answer should just be SUPPORTS, REFUTES, or NOT ENOUGH INFO and nothing else.

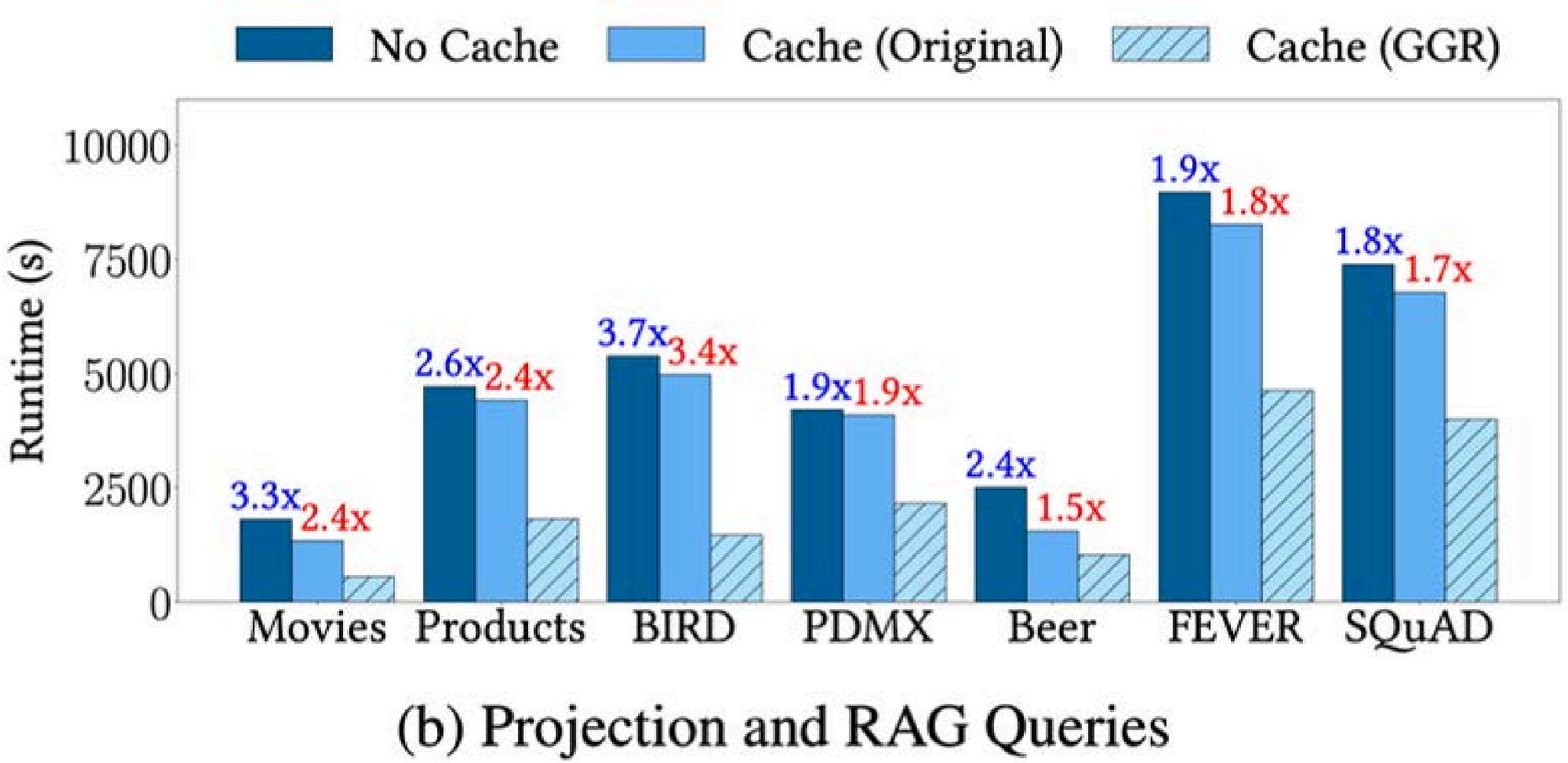
EVALUATION

Dataset	n_{rows}	n_{fields}	Tokens $\text{input}_{\text{avg}}$	Tokens $\text{output}_{\text{avg}}$	Query Type
Movies	15000	8	276	{2, 29, 16, 2}	T1-T4
Products	14890	8	377	{3, 107, 62, 2}	T1-T4
BIRD	14920	4	765	{2, 43}	T1, T2
PDMX	10000	57	738	{2, 72}	T1, T2
Beer	28479	8	156	{2, 38}	T1, T2
SQuAD	22665	5	1047	11	T5
FEVER	19929	5	1302	3	T5

EVALUATION


No Cache
Cache (Original)
Cache (GGR)
Runtime (s)
6000
4000
2000
0
3.8x
2.5x
3.8x
3.0x
2.7x
2.6x
2.1x
1.8x
3.8x
2.0x
Movies
Products
BIRD
PDMX
Beer
(a) Filter Queries

EVALUATION



(b) Projection and RAG Queries

EVALUATION

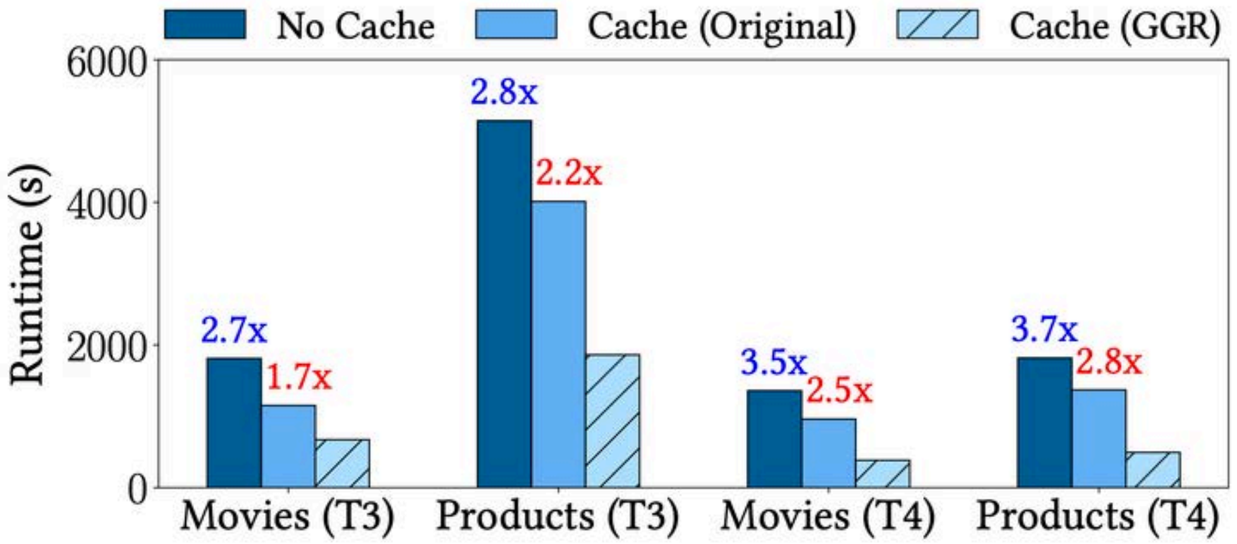


Figure 4. End-to-end Result (Multi-LLM Invocation, Aggregation): Our optimizations Cache (GGR) achieve 1.7 - 2.8× speed-up over Cache (Original), and 2.7 - 3.7× speed-up over No Cache.

EVALUATION

Method	Movies	Prods.	BIRD	PDMX	Beer	FEVER	SQuAD
Original	35%	27%	10%	12%	50%	11%	11%
GGR	86%	83%	85%	57%	80%	67%	70%

Table 2. PHR (%) of LLM Filter and RAG queries for Original and GGR, which achieves 30 – 75% higher hit rates.

EVALUATION

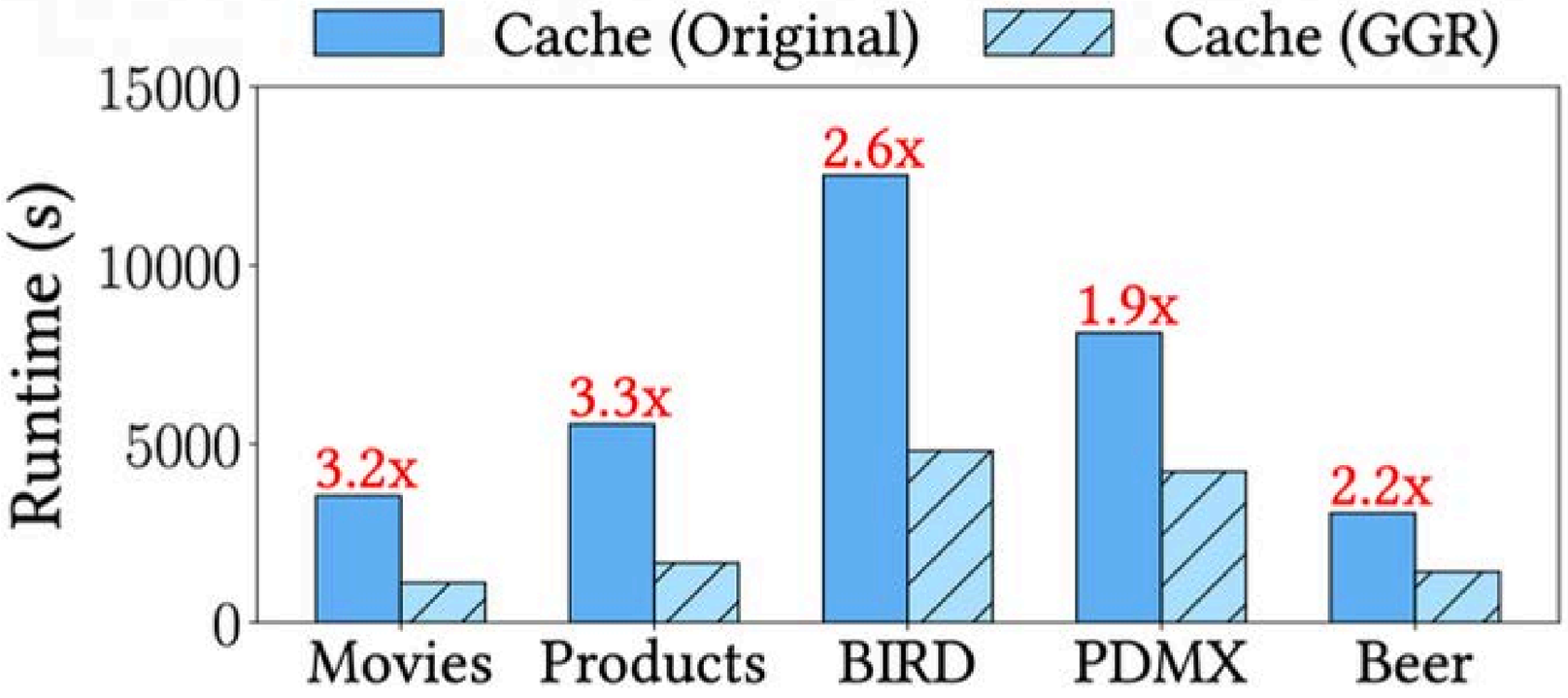


Figure 5. Cache (GGR) is able to achieve 1.9 – 3.3× speed-up over Cache (Original) for filter queries on Llama3-70B.

EVALUATION

Dataset	Model	Method	PHR (%)	Cost ($)	Savings (%)
FEVER	4o-mini	Original	0.0	0.81	-
		GGR	62.2	0.55	32%
	Sonnet	Original	0.0	5.49	-
		GGR	30.6	4.33	21%

Table 3. OpenAI and Anthropic Costs: cache hit rate (HR%), cost, and savings comparison of GGR over Original for GPT-4o-mini and Claude 3.5 Sonnet in FEVER.

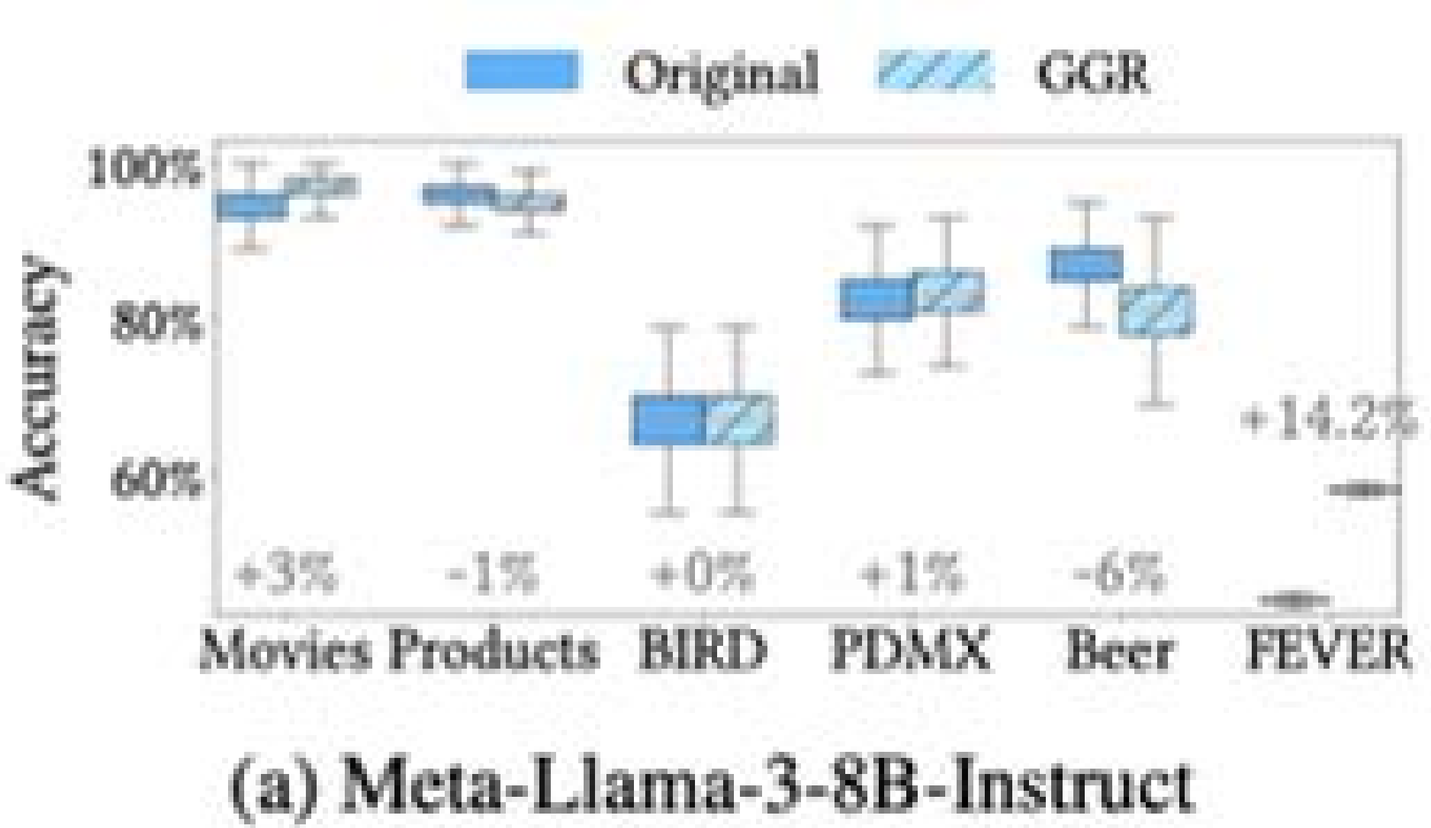
EVALUATION

Dataset	PHR (%)		Est. Cost Savings (%)	
	Original	GGR	OpenAI	Anthropic
Movies	34.6	85.7	31	73
Products	26.7	83.3	33	73
BIRD	10.4	84.8	39	79
PDMX	11.8	56.6	24	48
Beer	49.9	80.1	20	55
FEVER	11.2	67.4	30	60
SQuAD	11.0	69.7	31	63

Table 4. Estimated cost savings: across datasets using PHR from Sec 6.2 and OpenAI and Anthropic's pricing model.

Boston University

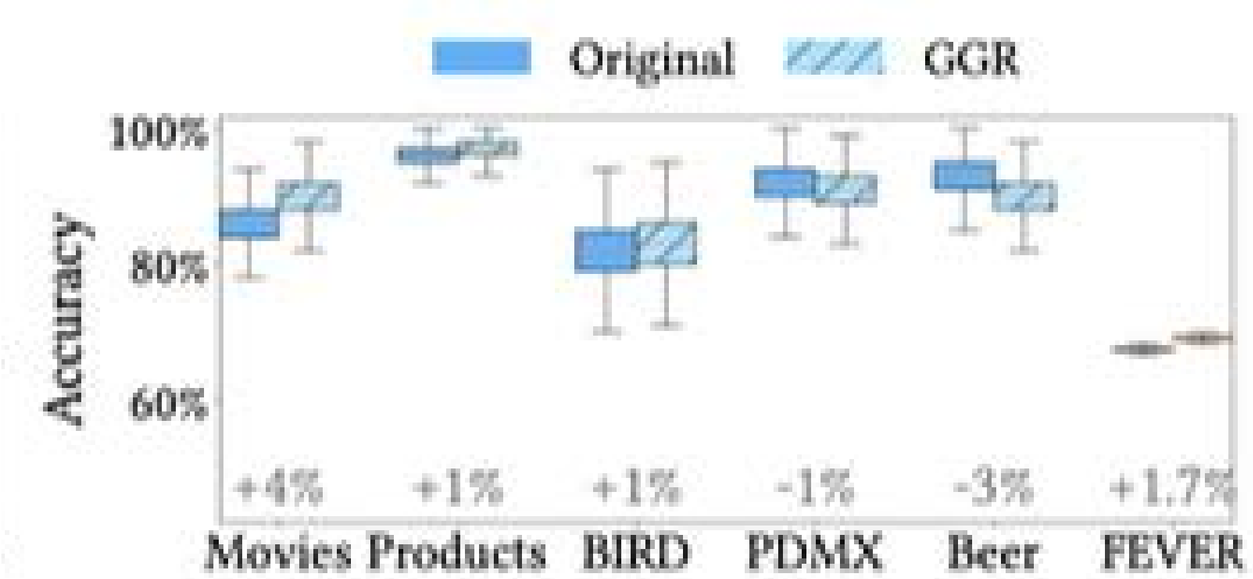
EVALUATION



(a) Meta-Llama-3-8B-Instruct

Boston University

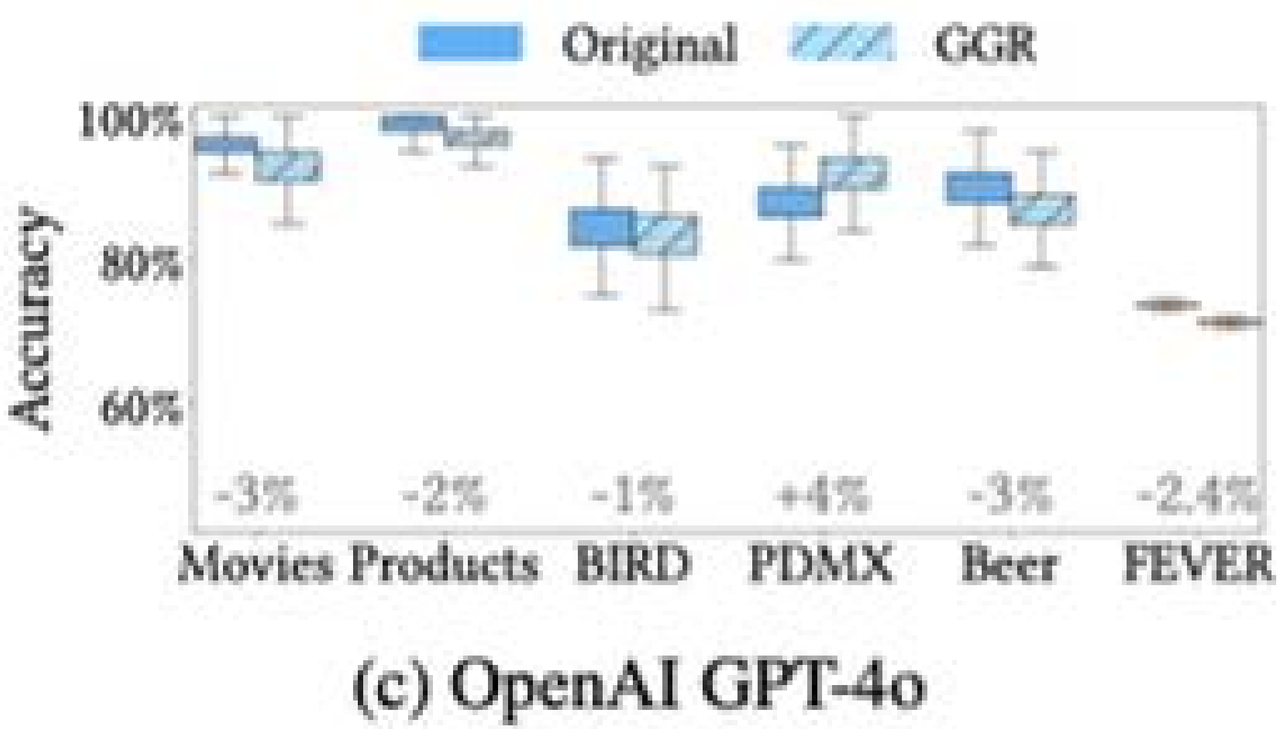
EVALUATION



(b) Meta-Llama-3-70B-Instruct

Boston University

EVALUATION



(c) OpenAI GPT-4o

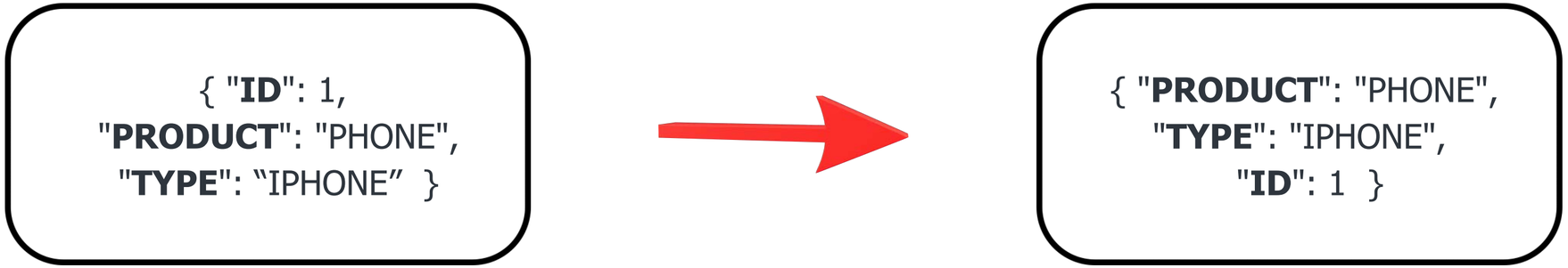
DISCUSSION

QUESTION 1: DOES FIELD REORDERING CHANGE LLM OUTPUTS?

- LLMs exhibit a "primacy effect," where fields appearing earlier in the prompt carry more weight in the model's output, so recording a row and moving a field from first to third position may change the answer it produces

- If reordering alters query results, this is no longer a pure optimization, it changes the output of the query itself, undermining the core claim of the paper

QUESTION 1: DOES FIELD REORDERING CHANGE LLM OUTPUTS?



Section 5 (Implementation)

- Rows are encoded in JSON
- Field–value mappings remain explicit
- Therefore, semantic meaning is preserved

(Some sensitivity may still remain)

QUESTION 2: ARE COST SAVINGS MEASURED ON REALISTIC DATA?

- The 32% cost saving was not measured on real data. Field values were duplicated 5x specifically to cross the token threshold that triggers API caching, a condition real datasets may never naturally reach

- If the prompts of a real workload fall below 1,024 tokens, caching never activates and GGR's reordering produces zero cost benefit regardless of how well it improves the theoretical hit rate

QUESTION 2: ARE COST SAVINGS MEASURED ON REALISTIC DATA?

Caching depends on prompt length

Below threshold → No caching → No cost savings

At / above threshold → Caching activated → Cost savings

Paper Setup:

- Prompts are artificially extended
- Caching threshold is reached

NOTE: ACTUAL COST SAVINGS DEPEND ON PROVIDER-SIDE CACHING THRESHOLDS

QUESTION 3: WHAT HAPPENS WITH LOW-REPETITION DATASETS?

- GGR's entire mechanism depends on finding repeated values to group; datasets with unique fields like timestamps, transaction IDs, or free-text notes offer nothing to reorder beneficially

QUESTION 3: WHAT HAPPENS WITH LOW-REPETITION DATASETS?

When is GGR most effective?

High Repetition	Low Repetition
High cache reuse	Low cache reuse
Less recomputation	More recomputation
Higher optimization gains	Limited/low optimization gains

MORE REPETITION → MORE PREFIX REUSE → HIGHER OPTIMIZATION GAINS

QUESTION 4: THE EVALUATION USES A SINGLE QUERY AT A TIME; WHAT HAPPENS UNDER CONCURRENT LOAD?

- All experiments were run in isolation with the KV cache exclusively dedicated to one query, meaning no competing requests are evicting the carefully arranged prefixes GGR depends on

- In any real multi-tenant deployment, concurrent jobs compete for the same cache space, and the speedup numbers the paper reports could be substantially optimistic as a result

QUESTION 4: THE EVALUATION USES A SINGLE QUERY AT A TIME; WHAT HAPPENS UNDER CONCURRENT LOAD?

GGR's benefit is not only algorithm-dependent, but also system-dependent

SHARED/GLOBAL CACHE

Q1 → [PREFIX A]

Q2 → [PREFIX B]

Q3 → [PREFIX C]

➡ **PREFIX A EVICTED BEFORE REUSE**

ISOLATED CACHE

USER GROUP 1 → CACHE WORKER 1 → [PREFIX A REUSED]

USER GROUP 1 → CACHE WORKER 2 → [PREFIX B REUSED]

➡ **CACHE LOCALITY PRESERVED**

THANK YOU