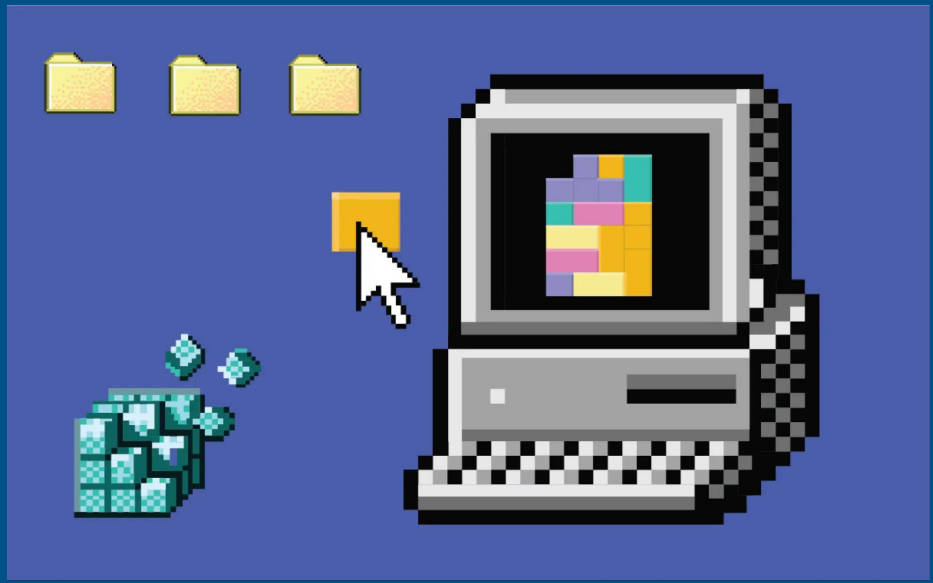
April 2025

Selection Pushdown in Column Stores

Using Bit Manipulation Instructions

Linfeng Zhu, Haonan Wu, Anastas Kanaris, Bruce Casillas, Minhong Zou, Rohit Saha



1. Introduction & Background
2. Bit Parallel Select Operator
3. Selection Pushdown
4. Selection Pushdown in Parquet
5. Evaluation & Conclusion

1. **Introduction & Background**
2. Bit Parallel Select Operator
3. Selection Pushdown
4. Selection Pushdown in Parquet
5. Evaluation & Conclusion

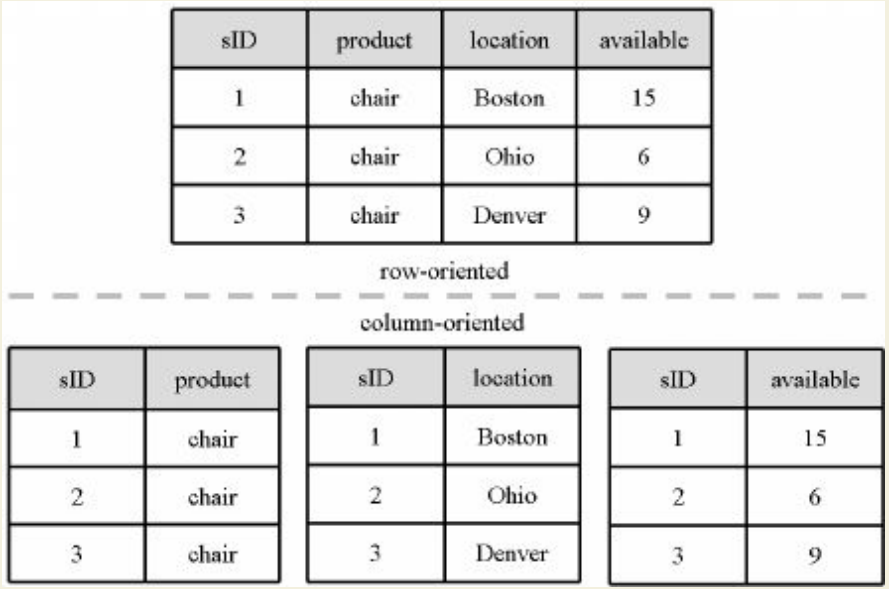
Column-oriented vs Row-oriented

Row-Oriented Storage

- Stores data **row by row** (e.g., traditional relational databases).
- Efficient for **transactional workloads** (OLTP) with frequent inserts/updates.
- Fast access to **all fields of a single record**.

Column-Oriented Storage

- Stores data **column by column** (e.g., Parquet).
- Optimized for **analytical workloads** (OLAP) with large scans and aggregations.
- Better **compression** and **I/O efficiency** for selected columns.
- Enables advanced techniques like **predicate pushdown** and **vectorized processing**.



sID	product	location	available
1	chair	Boston	15
2	chair	Ohio	6
3	chair	Denver	9

row-oriented

column-oriented

sID	product
1	chair
2	chair
3	chair

sID	location
1	Boston
2	Ohio
3	Denver

sID	available
1	15
2	6
3	9

Compression and Decoding Challenges in Column Stores

Compression via Dictionary Encoding

- Columnar layout → similar consecutive values → **high compressibility**.
- **Dictionary encoding** maps each unique value to a **small code**.
- Codes stored in a **bit-packed format** using minimal bits.

Decoding Bottlenecks

- Queries must **decode** these codes before processing.
- Even with **SIMD-based decoding**, performance is limited:
  - Decoded values (e.g., 64-bit ints) are much **larger** than encoded bits.

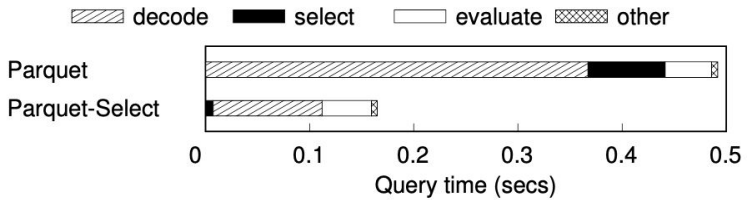


Fig. 1. Time breakdown of TPC-H Q6

Decoding dominates query time!

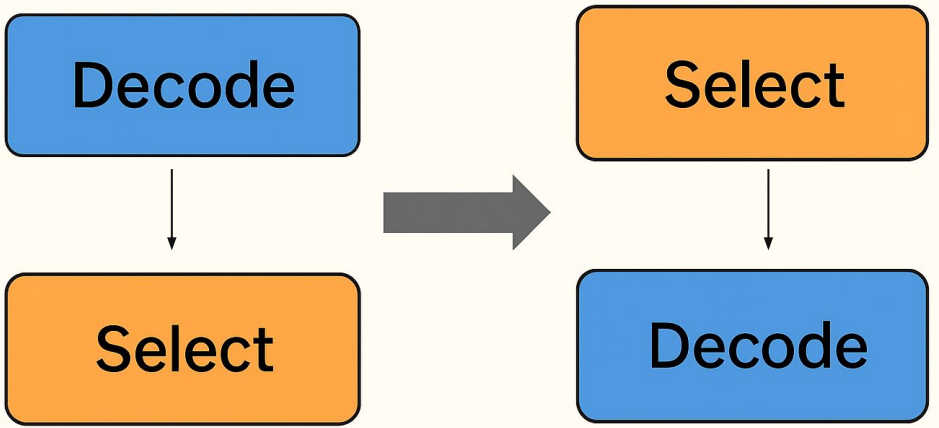
Predicate Pushdowns

Traditional Predicate P

- Applies filters direct
- Requires:
  1. Encoding to
  2. Predicates t

Limitations

- Not compatible with
  not order-preservin
- Fails to support con
  - String match
  - User-define
  - Cross-colum



n Pushdown

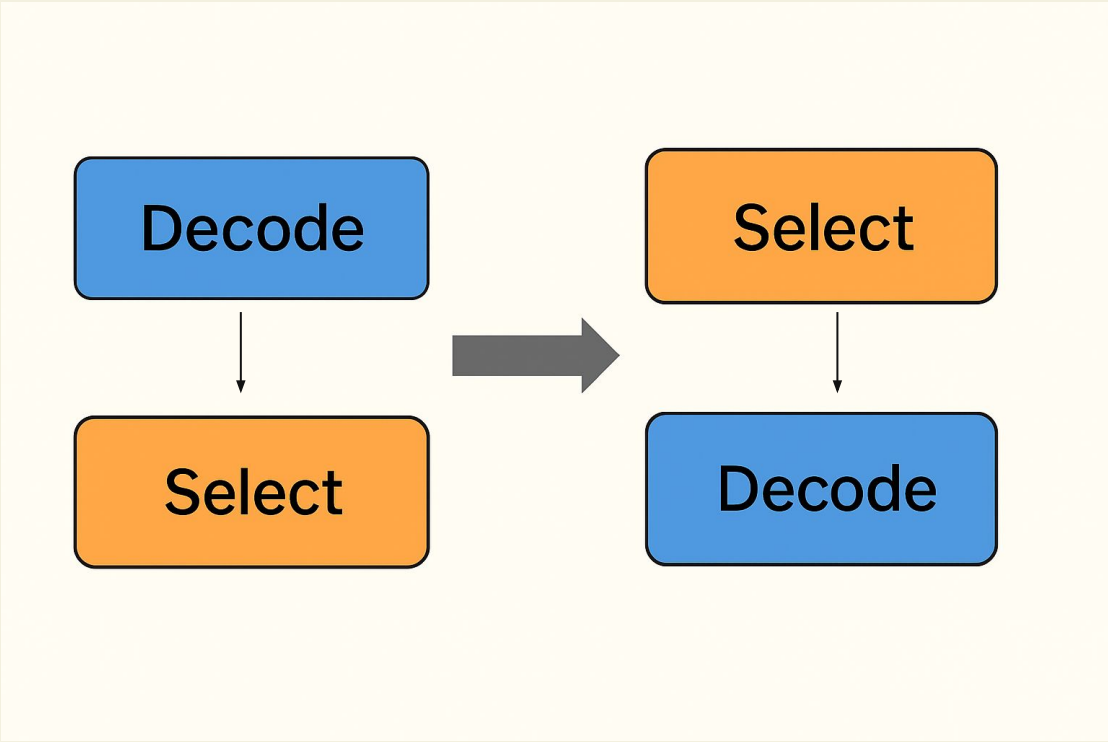
sequentially, leveraging
ilters.

rator to filter encoded
oding.

oded values are
uated.

predicates, even with
ing encodings.

Predicate Pushdowns



The Challenge of Efficient Selection on Encoded Data

❗ The Problem

- Goal: Design a fast **select operator** that works directly on **bit-packed encoded values**.
- Needs to move all selected values simultaneously from a processor word.

❗ Why It's Hard

- Extremely difficult without specialized hardware support.
- Standard software techniques are **too slow or inefficient** for this task.

⭕ The Key Insight

- **Bit Manipulation Instructions (BMI)**, introduced by Intel in 2013, provide the necessary capabilities.
- Now widely supported in Intel and AMD processors.

✓ Novel Contribution

- This paper is the **first to apply BMI** to database systems in this context.
- Designing an efficient BMI-based operator is **nontrivial**, which likely explains why it has been overlooked until now.

The Broader Role of BMI in Selection Pushdown

Beyond the Select Operator

- **Select Operator**
  - Extracts selected valu
  using a select bitmap
  - BMI enables fast sele
  bit-widths using only
  word.
- **Selection Pushdown Fram**

Implementation: Parquet-Select

- A lightweight library built on these techniques.
- Compatible with standard Parquet files without changing the format.
- Evaluation shows:
  - Up to 10x speedup on scan queries.
  - Up to 5.5x speedup in Spark for end-to-end complex queries.

General Applicability

- Techniques are applicable to other formats like Apache ORC, Arrow, and custom internal formats.

omplex Structures

s Parquet's nested and repeated
th structural metadata
on/definition levels).
sed to transform bitmaps and
t structure-encoding integers.

Applies filter & project operations using BMI to pre-select encoded values

Efficiently transforms bitmaps to align with predicate evaluations

Reduces decoding and selection costs across scan queries

Background

Bit Manipulation Instructions: PEXT and PDEP

Overview of BMI

- An extension to the **x86 architecture** for Intel and AMD CPUs.
- Designed to speed up **bitwise operations** using dedicated hardware instructions.
- Operates on **64-bit general-purpose registers**, unlike SIMD which uses vector registers.
- Consists of **14 instructions**, many replacing common software-implemented patterns.

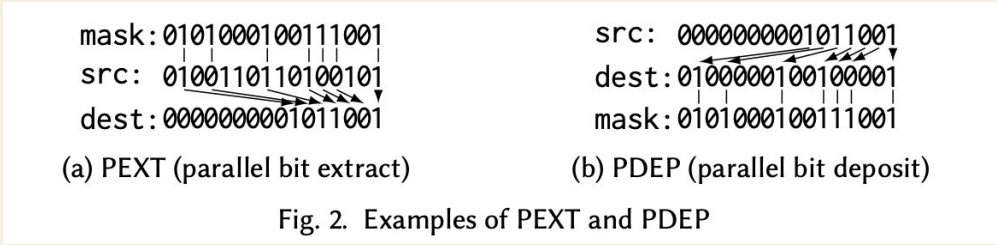


Fig. 2. Examples of PEXT and PDEP

Focus: PEXT and PDEP

- **PEXT (Parallel Bit Extract)**:
  Extracts bits from a source based on a mask and packs them into the low-order bits of the result.
- **PDEP (Parallel Bit Deposit)**:
  Takes low-order bits from a source and distributes them into a destination according to a mask.

Bit Manipulation Instructions: PEXT and PDEP

Overview of BMI

- An extension to the **x86 architecture** for Intel and AMD
-
-
-

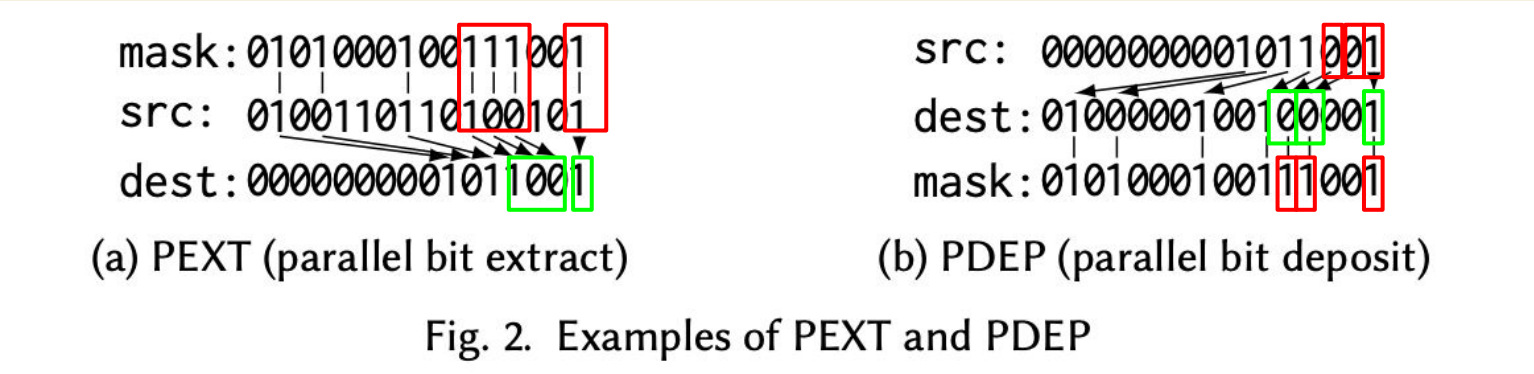


Fig. 2. Examples of PEXT and PDEP

Focus:

- **PEXT (Parallel Bit Extract)**:
  Extracts bits from a source based on a mask and packs them into the low-order bits of the result.
- **PDEP (Parallel Bit Deposit)**:
  Takes low-order bits from a source and distributes them into a destination according to a mask.

BMI vs Software Implementation

Throughput (ops/s)	Intel Xeon Gold 6140			AMD EPYC 7413		
	BLSI	PEXT	PDEP	BLSI	PEXT	PDEP
Software	3100M	8.1M	8.7M	6214M	18.3M	18.5M
BMI	1381M	1150M	1143M	1243M	1713M	1651M
Speedup	**0.46X**	**142X**	**131X**	**0.2X**	**94X**	**89X**

Table 1. BMI vs. software implementation

Apache Parquet: Columnar Storage Design

Overview

- Open-source columnar storage format based on Google's **Dremel**.
- Widely adopted in the **Big Data ecosystem** for efficient analytics.

Data Model & Schema

- Inherits schema from **Protocol Buffers**.
- Supports **strongly-typed nested structures**:
  - Fields can be **atomic** or **group (nested)**.
  - Each field has a **data type** and a **repetition type**:
    - `required`, `optional`, or `repeated`

Repetition & Definition Levels

- Used to represent **nulls** and **repeated structures**.
- Stored as two small integers for each value.
- Help reconstruct the original tree-like record structure.

Encoding

- Parquet uses a **hybrid encoding** scheme:
  - Combines **Run-Length Encoding (RLE)** and **Bit-Packing**.
  - Dictionary encoding maps values to codes (not order-preserving).
  - Falls back to **plain encoding** if dictionary grows too large.
- Structural metadata (repetition/definition levels) also encoded using RLE/bit-packing.

Apache Parquet: Columnar Storage Design

Record	phones.number
1	["123", null, "456"]
2	null

Row	Repetition Level	Definition Level	Value
1	0	2	"123"
2	1	1	null
3	1	2	"456"
4	0	0	null

Repetition & Definition Levels

- Used to represent **nulls** and **repeated structures**.
- Stored as two small integers for each value.
- Help reconstruct the original tree-like record structure.

Encoding

- Parquet uses a **hybrid encoding** scheme:
  - Combines **Run-Length Encoding (RLE)** and **Bit-Packing**.
  - Dictionary encoding maps values to codes (not order-preserving).
  - Falls back to **plain encoding** if dictionary grows too large.
- Structural metadata (repetition/definition levels) also encoded using RLE/bit-packing.

Apache Parquet: Columnar Storage Design

```
["Apple", "Apple", "Apple", "Banana", "Cherry", "Cherry", "Cherry", "Cherry"]
```

Dictionary:

```
0 → Apple
1 → Banana
2 → Cherry
```

```
[0, 0, 0, 1, 2, 2, 2, 2]
```

RLE:

```
0 repeated 3 times → RLE(0, 3)
2 repeated 4 times → RLE(2, 4)
```

```
[RLE: 0 × 3], [Bit-Packed: 1], [RLE: 2 × 4]
```

Apache Parquet: Columnar Storage Design (cont'd)

Storage Format

- Data is partitioned into **row groups** (row-major).
- Within each row group: **column-major** layout (like PAX).
- Each column includes:
  - **Field values**
  - **Repetition levels**
  - **Definition levels**

- Optimizations:


Null values not stored


Definition levels omitted for required fields

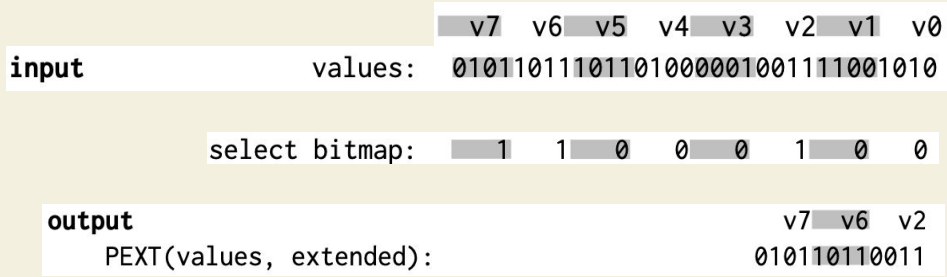

Repetition levels omitted for non-repeated fields

1. Introduction & Background
2. **Bit Parallel Select Operator**
3. Selection Pushdown
4. Selection Pushdown in Parquet
5. Evaluation & Conclusion

Bit Parallel Algorithm: Problem and Motivation

Problem Statement

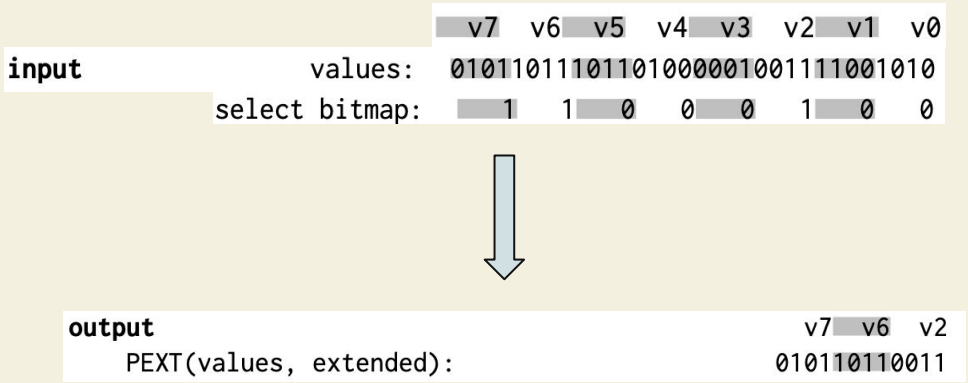
- **Input**:
  - A byte array with n values, each k bits wide (bit-packed).
  - An n-bit **select bitmap** indicating which values to extract.
- **Goal**:
  Extract all values where the corresponding bit in the bitmap is 1, and pack them **contiguously** in the output.



Example Case

- Selecting 3 out of 8 **4-bit** values: output includes only the selected values.
- Complex case: selecting **3-bit** values across 32-bit word boundaries, making alignment and extraction harder.

Bit Parallel Algorithm: Problem and Motivation



Naïve Approach

- Scan through the array and extract selected values one by one:
  - Takes `O(n)` instructions.
  - **Inefficient**: fails to exploit the full width of modern CPUs (e.g., 64-bit words).

Goal: Bit-Parallel Select Operator

- **Bit-parallel** means: process all values in a processor word **in parallel**, not sequentially.

- This enables **higher throughput** by leveraging CPU word-level parallelism.

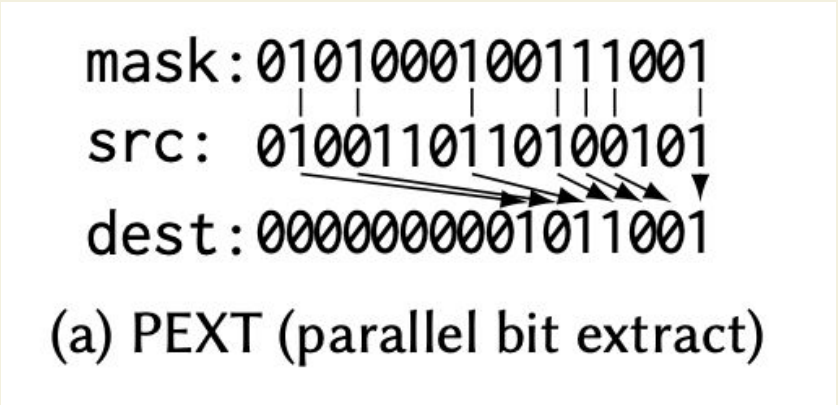
Simplified Bit-Parallel Selection Algorithm

Assumptions

- Bit width `k` is a **power of 2** (e.g., 1, 2, 4, 8).
- No value crosses processor word boundaries.

Special Case: `k = 1`

- Goal: Extract all bits in the value array where the **select bitmap** has 1s.
- **Solution**: Use **PEXT** directly:
  - `values` → source operand
  - `select bitmap` → mask operand
  - PEXT extracts matching bits in one instruction.

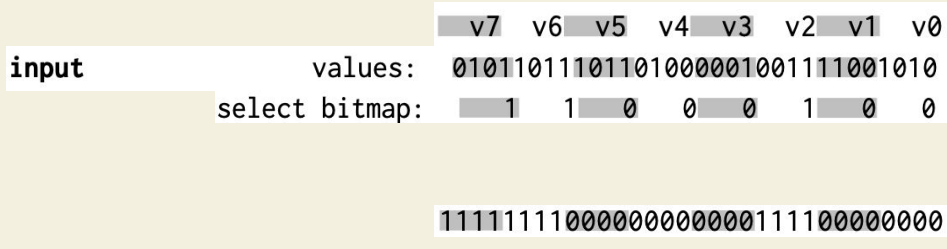


(a) PEXT (parallel bit extract)

Fast Select Operator: Problem and Motivation

Generalization to k-bit Values

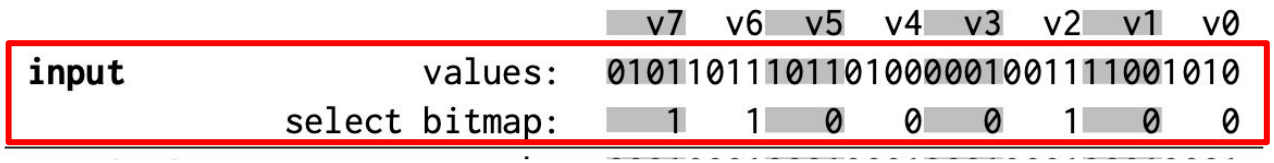
- Direct PEXT won't work — need to extract **k bits** for each selected value.
- **Solution**:
  - Create an **extended bitmap** by duplicating each bit in the select bitmap `k` times.
  - Example: select bitmap `11000100`, `k = 4` → extended bitmap becomes `11111111000000000000111100000000`.



Algorithm Steps

1. **Step 1**: Generate the extended bitmap from the original select bitmap.

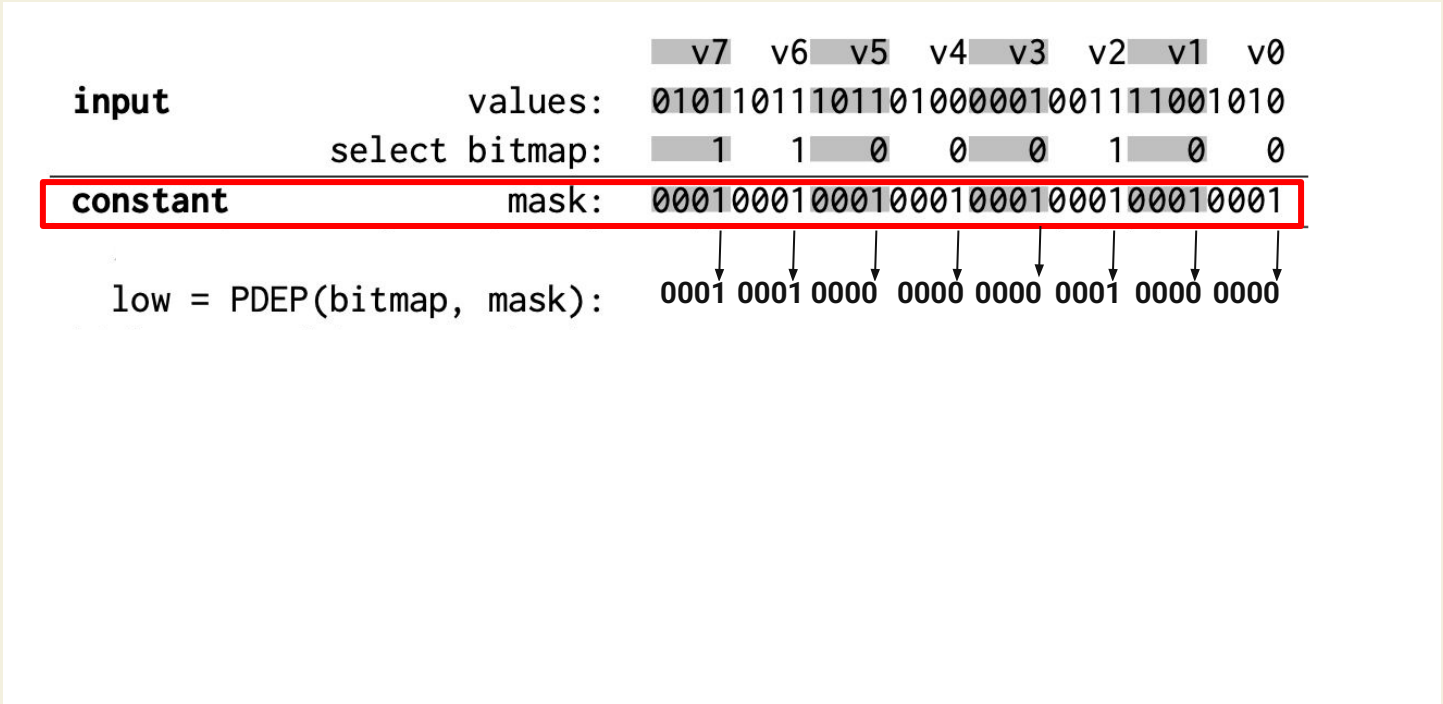
2. **Step 2**: Use **PEXT** with the extended bitmap to extract all selected k-bit values into contiguous output space.



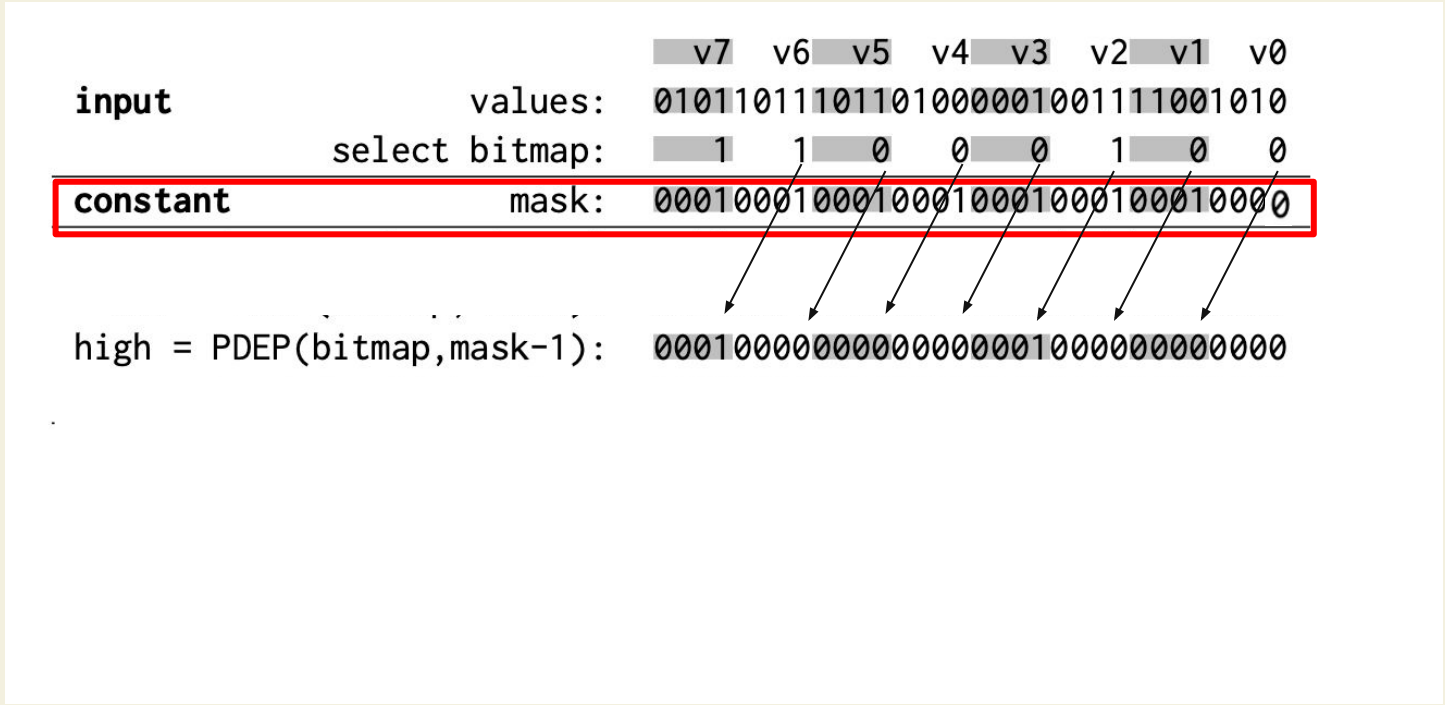
! We need to extend the select bitmap

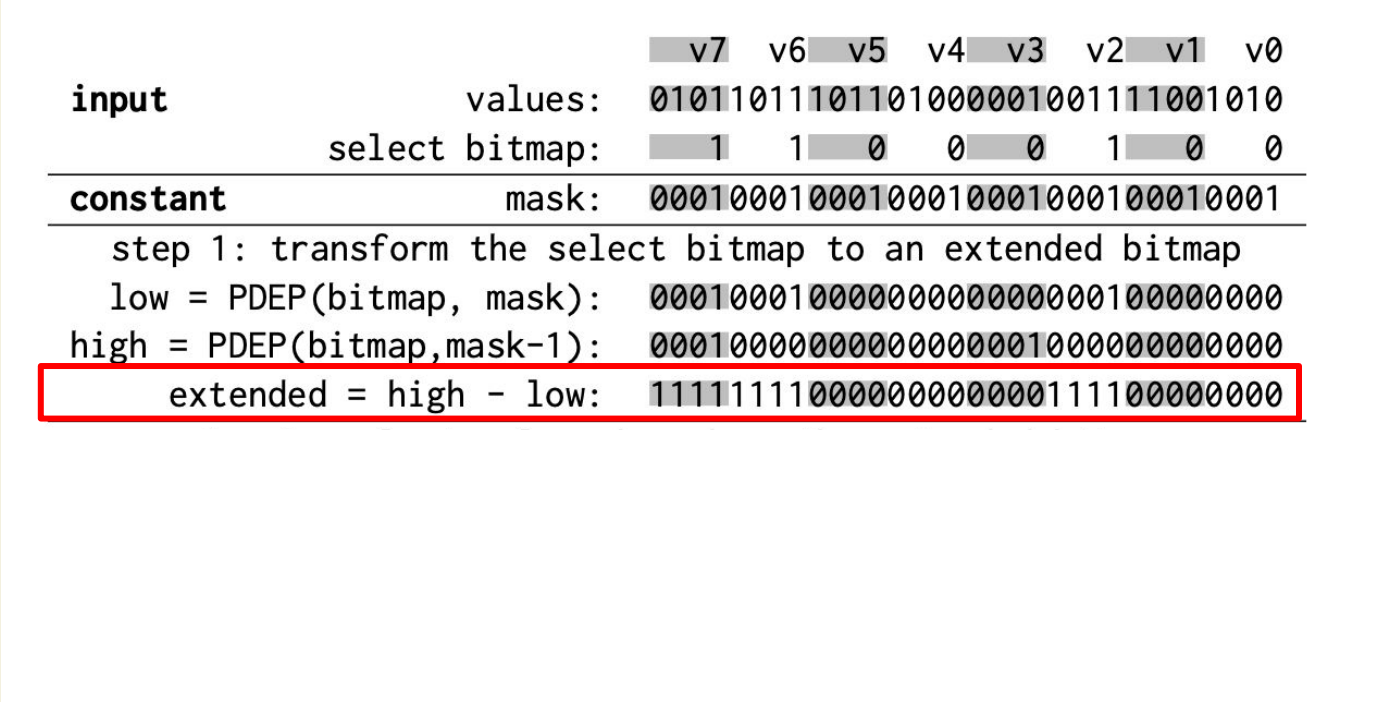
		v7	v6	v5	v4	v3	v2	v1	v0
input	values:	01011011	10110100	00010011	11001010				
	select bitmap:	1	1	0	0	0	1	0	0
constant	mask:	00010001	00010001	00010001	00010001				

$$mask = 0^{k-1}1 \ldots 0^{k-1}1$$


v7 v6 v5 v4 v3 v2 v1 v0
input values: 01011011101101000010011110010 10
select bitmap: 1 1 0 0 0 1 0 0
constant mask: 0001000100010001000100010001 0001
low = PDEP(bitmap, mask): 0001 0001 0000 0000 0000 0001 0000 0000

Bit Parallel Select Operator




v7 v6 v5 v4 v3 v2 v1 v0
input values: 01011011101101000010011110 01010
select bitmap: 1 1 0 0 0 1 0 0
constant mask: 00010001000100010001000100010001
step 1: transform the select bitmap to an extended bitmap
low = PDEP(bitmap, mask): 00010001000000000000000100000000
high = PDEP(bitmap,mask-1): 00010000000000000001000000000000
extended = high - low: 11111111000000000000111100000000

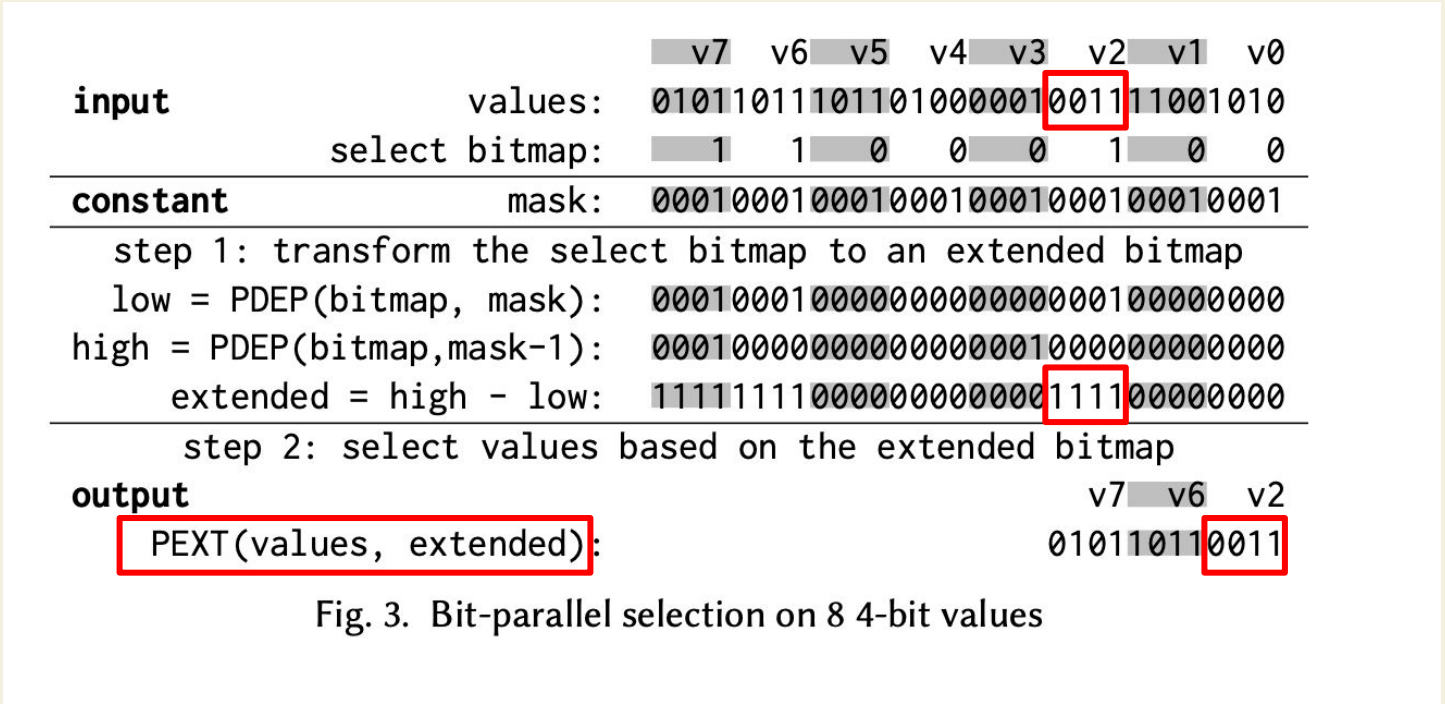


Fig. 3. Bit-parallel selection on 8 4-bit values

Algorithm 1 select (*values*, *bitmap*, *mask*)

1: *extended* := extend(*bitmap*, *mask*)
2: **return** PEXT(*values*, *extended*)

Algorithm 2 extend (*bitmap*, *mask*)

1: *low* := PDEP(*bitmap*, *mask*)
2: *high* := PDEP(*bitmap*, *mask* - 1)
3: **return** *high* - *low*

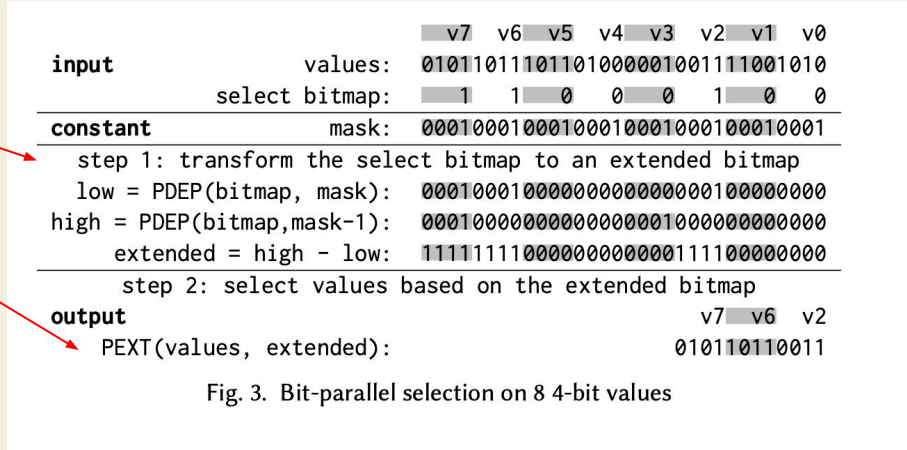


Fig. 3. Bit-parallel selection on 8 4-bit values

What if k is not a multiple of 2?

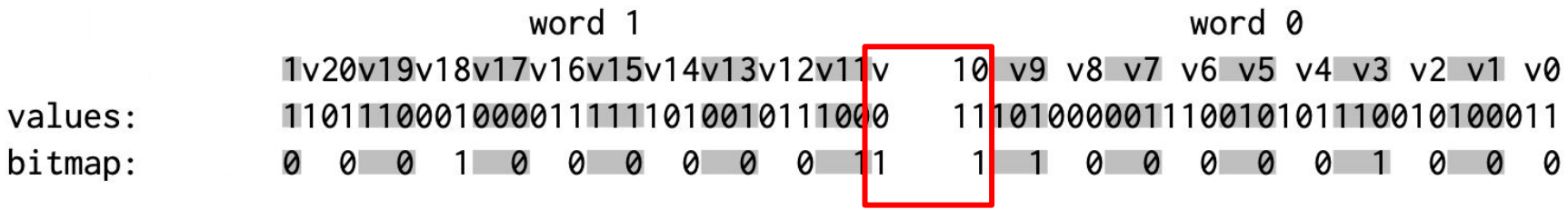


Fig. 4. Bit-parallel selection on 32 3-bit values (v10 and v21 span over multiple words)

General Algorithm

Key Insight

- **Algorithm 1** still works with **partial values** in a processor word (i.e., values spanning word boundaries).
- Valid as long as **two key conditions** on the mask are met.

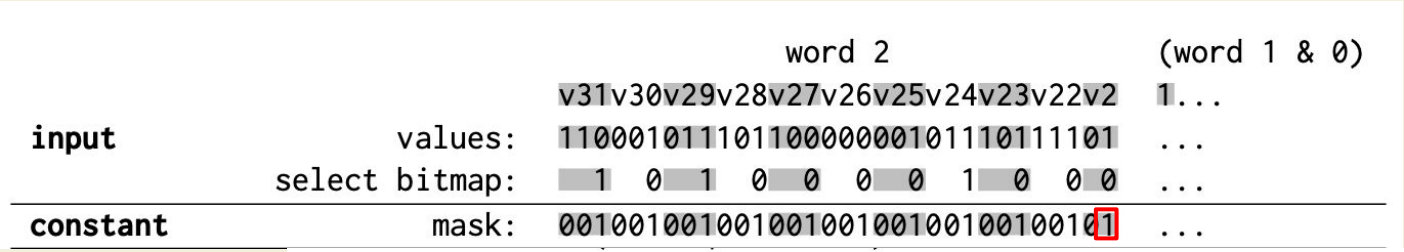
Condition 1: Mask Alignment

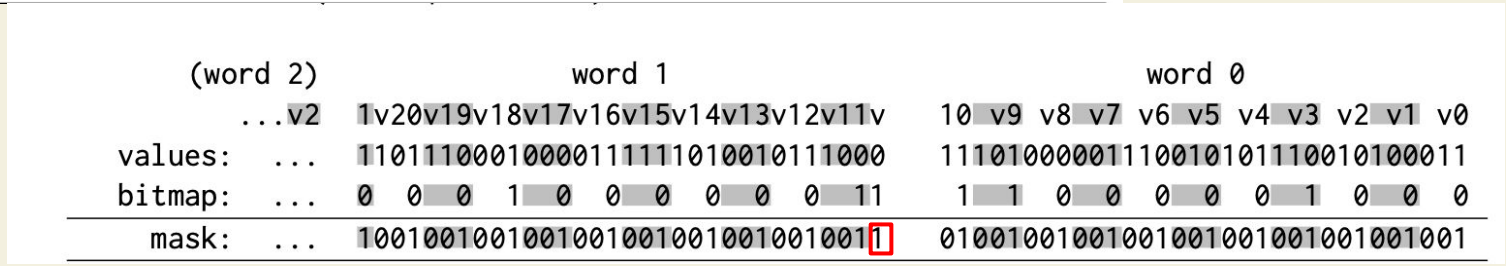
- The **mask must be left-shifted** to align with the layout of the word.

Condition 2: Least Significant Bit (LSB) = 1

- The LSB of the mask must be set to **1**, even if it's in the middle of a value.
- Ensures that the **subtraction instruction** generates a correct 1s run in the extended bitmap.

General Algorithm





1. Introduction & Background
2. Bit Parallel Select Operator
3. **Selection Pushdown**
4. Selection Pushdown in Parquet
5. Evaluation & Conclusion

Selection Pushdown

Goal

- Accelerate **arbitrary scan queries** using the **fast BMI-based select operator**.

Query Model

- Focus on queries with:
  - **Projection columns** (`SELECT`)
  - **Filter columns** (`WHERE`)
- Initially assumes a **conjunction of filters** (to be extended later to general boolean expressions).

```
SELECT c FROM R WHERE a < 10 AND b < 4
```

Key Observation

- **Bypass records** that fail prior predicates.

- Previous work **evaluates all predicates** for all records, even if already filtered out.
- This was due to **high overhead** of selection.

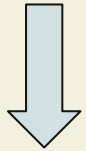
Selection Pushdown

BMI-Based Fast Select Operator

- Leverage the **fast, BMI-enabled select operator** to:
  - **Select values upfront**
  - Apply it in both **filter** and **project** phases
- This approach makes **select-first filtering efficient and practical**.

```
SELECT c FROM R WHERE a < 10 AND b < 4
```

```
SELECT c FROM R WHERE a < 10 AND b < 4
```

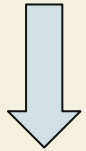


column a bitmap_a = filter(a, null, < 10)

bitmap_a: 1010000100000100000011100000 1000

Fig. 5. Operations in evaluating the example query

```
SELECT c FROM R WHERE a < 10 AND b < 4
```



column a	$bitmap_a$ = filter(a, null, < 10)
	$bitmap_a$: 1010000100000100000011100000 01000
column b	$bitmap_b$ = filter(b, $bitmap_a$, < 4)
	$bitmap_b$: 0010000100000100000001000000 0000
column c	$selected_c$ = project(c, $bitmap_b$)
	$selected_c$: v29v24v18v10

Fig. 5. Operations in evaluating the example query

1. Select

- Applies the **BMI-based select operator** to filter out irrelevant values early.
- Reduces the number of values passed to later operators.
- Can be **skipped** for the first filter

2. Unpack

- Converts encoded values into **primitive data types**.
- Uses SIMD-based decoding for performance.
- **Final step for project operations**—no need to evaluate or transform afterward.

3. Evaluate

- Applies the **filter predicate** on decoded values.
- Produces a **bitmap** indicating which selected values satisfy the predicate.
- Enables **arbitrary predicates** and leverages **SIMD vectorization**.

4. Transform

- Converts the bitmap (from `Evaluate`) into a **select bitmap** usable by the next step.
- Necessary because the evaluate bitmap only covers filtered values.
- Efficiently implemented using **BMI techniques**.

column a	filter(a, null, < 10) = evaluate$_{<10}$(unpack(a))

Table 2. Implementation of example filter and project operations

column a	filter(a, null, < 10) = evaluate$_{<10}$(unpack(a))
column b	filter(b, bitmap$_a$, < 4) = transform(evaluate$_{<4}$(unpack(select(b, bitmap$_a$))), bitmap$_a$)

Table 2. Implementation of example filter and project operations

column a	filter(a, null, < 10) = evaluate$_{<10}$(unpack(a))
column b	filter(b, bitmap$_a$, < 4) = transform(evaluate$_{<4}$(unpack(select(b, bitmap$_a$))), bitmap$_a$)
column c	project(c, bitmap$_b$) = unpack(select(c, bitmap$_b$))

Table 2. Implementation of example filter and project operations

1. Introduction & Background
2. Bit Parallel Select Operator
3. Selection Pushdown
4. **Selection Pushdown in Parquet**
5. Evaluation & Conclusion

Overview

- In Parquet, each column value is represented as a triple:

  ⟨repetition level, definition level, field value⟩

```
[
  { "user": { "name": "Alice", "scores": [80, 90] } },
  { "user": { "name": null, "scores": [] } },
  { "user": null }
]
```

	field values:	**definition levels**:	**repetition levels**:
Column: user.name	["Alice"]	[2, 1, 0]	[0, 0, 0]
Column: user.scores	[80, 90]	[3, 3]	[0, 1]

- Two key facts:

  1. A value is **null** if its **definition level < max definition level**

  2. A value **belongs to the same record** as the previous one if its **repetition level ≠ 0**

- **Key challenge**:

The number of values ≠ number of records → **select bitmap cannot be directly applied to column values**

- Therefore, we need to:

  Transform the input select bitmap into bitmaps for both field values and levels

Workflow

- A **select operation** takes:
  - Encoded repetition/definition levels
  - Field values
  - A **record-level select bitmap**
  - → returns the matching structural values

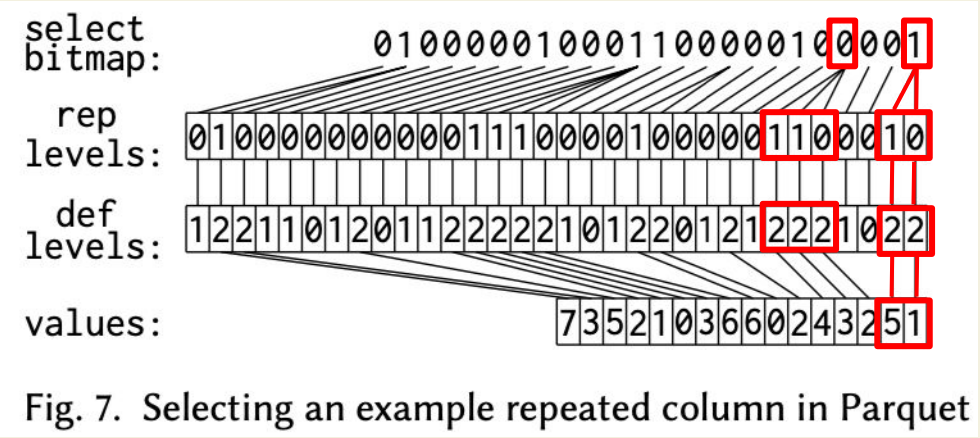


Fig. 7. Selecting an example repeated column in Parquet

- **Repetition levels** define record boundaries:
  - A new record starts when **repetition level = 0**
  - Values with repetition level ≠ 0 belong to the **same record**
- **Definition levels** indicate nulls:
  - Max definition level = 2, values with definition level < 2 are **null**
- A **24-bit select bitmap** is used:
  - Each bit represents whether a **record** is selected
- **Key insight**:
  - A value is selected **if and only if** it is **connected to a 1** in the select bitmap

Workflow

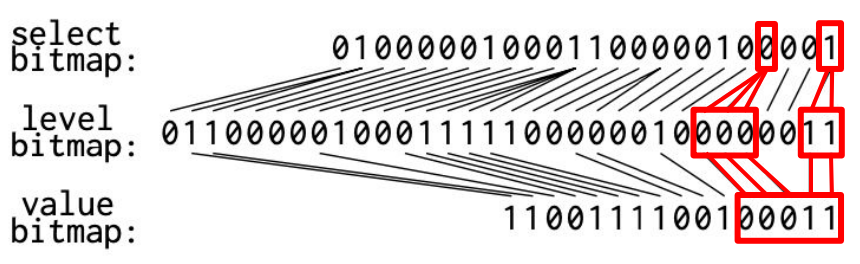
- The basic idea

Transform the input select bitmap to:

level bitmap and value bitmap

- The level bitmap is generated by copying each bit in the select bitmap as many times as the number of values in the corresponding record.

- The value bitmap can be created by removing the bits corresponding to null values from the level bitmap.

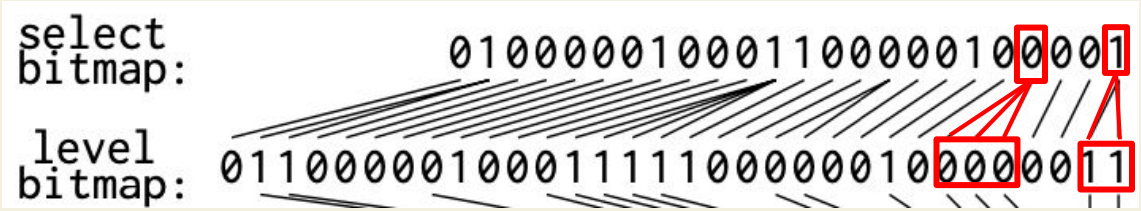


Algorithm 4 select-parquet $(reps, defs, values, b_{select})$

1: $b_{level} := b_{select}$
2: $selected\_reps := \emptyset$
3: **if** $reps \neq \emptyset$ **then**
4: $\quad b_{record} := \text{equal}(reps, 0)$
5: $\quad b_{level} := \text{extend}(b_{select}, b_{record})$
6: $\quad selected\_reps := \text{select}(reps, b_{level})$
7: $b_{value} := b_{level}$
8: $selected\_defs := \emptyset$
9: **if** $defs \neq \emptyset$ **then**
10: $\quad b_{valid} := \text{equal}(defs, max\_def\_level)$
11: $\quad b_{value} := \text{compress}(b_{level}, b_{valid})$
12: $\quad selected\_defs := \text{select}(defs, b_{level})$
13: $selected\_values := \text{select}(values, b_{value})$
14: **return** $\langle selected\_reps, selected\_defs, selected\_values \rangle$

Select Bitmap to Level Bitmap

1: $b_{level} := b_{select}$
2: $selected\_reps := \emptyset$
3: **if** $reps \neq \emptyset$ **then**
4: $b_{record} := \text{equal}(reps, 0)$
5: $b_{level} := \text{extend}(b_{select}, b_{record})$
6: $selected\_reps := \text{select}(reps, b_{level})$



- Goal:

For each 1 in the select bitmap, expand it to k 1s

For each 0, expand it to k 0s.

Equal operator

$b_{record} := \text{equal}(reps, 0)$

The Equal operator can determine whether each small integer is equal to a constant in a compressed state and quickly output a bitmap.

Algorithm 5 equal (*values*, *literal*)

1: $mask_{low} := 0^{k-1}1 \ldots 0^{k-1}1$
2: $mask_{high} := 10^{k-1} \ldots 10^{k-1}$
3: $literals := mask_{low} \times literal$
4: $results := \neg((values \oplus literals) \vee$
5: $(((values \oplus literals) \vee mask_{high}) - mask_{low}))$
6: **return** PEXT($results, mask_{high}$)

input	values (v):	0100011010000110011010100100 1010
	literals (l):	1010101010101010101010101010 1010
constant	$mask_{low}$ (lo):	0101010101010101010101010101 0101
	$mask_{high}$ (hi):	1010101010101010101010101010 1010
results = ¬((v⊕l)∨(((v⊕l)∨hi)-lo)):		0000001010000010001010100000 1010
output	b_{valid} = PEXT(results,hi):	0001100101110011

Fig. 9. Equality comparisons on 16 2-bit definition levels

Extend operator

Extend the select bitmap by using the record bitmap as the mask of the extend operator, duplicating each bit k times where k is the number of values in the corresponding record.

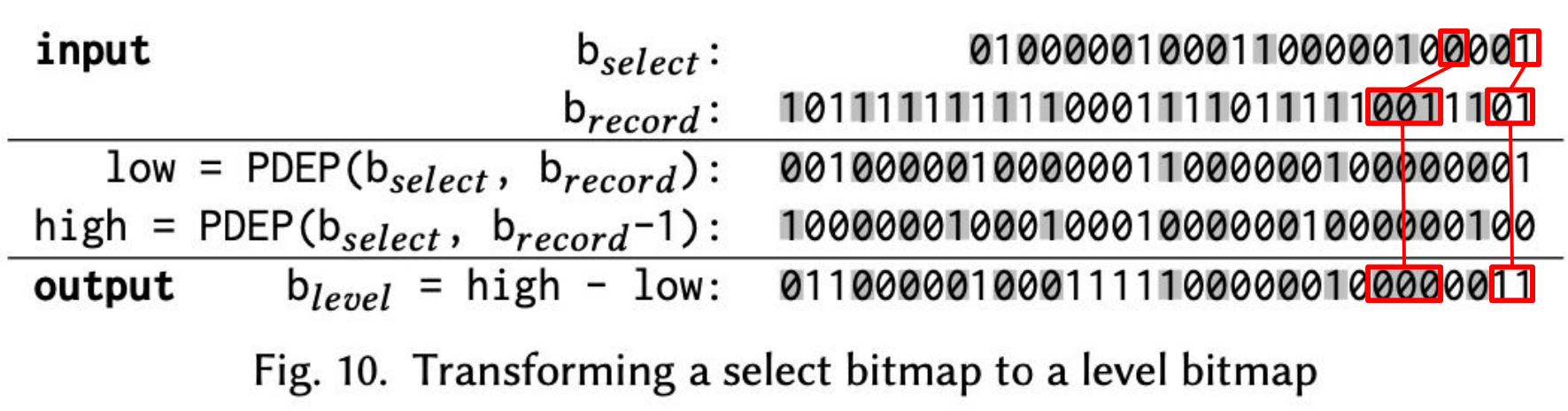
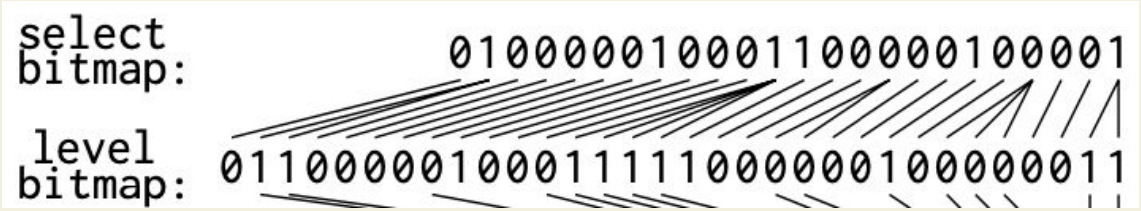


Fig. 10. Transforming a select bitmap to a level bitmap

Select Bitmap to Level Bitmap

1: $b_{level} := b_{select}$
2: $selected\_reps := \emptyset$
3: **if** $reps \neq \emptyset$ **then**
4: $\quad b_{record} := \text{equal}(reps, 0)$
5: $\quad b_{level} := \text{extend}(b_{select}, b_{record})$
6: $\quad selected\_reps := \text{select}(reps, b_{level})$



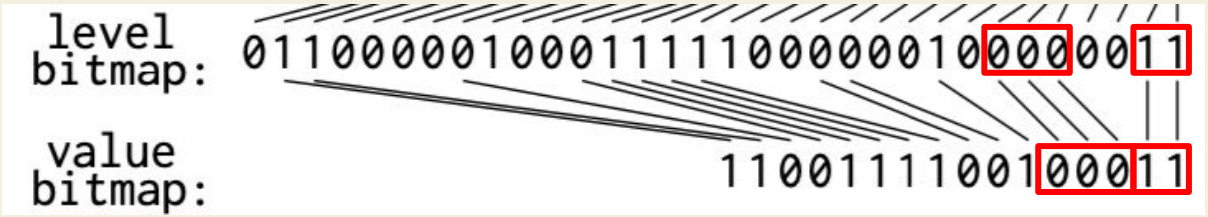
- To transform from a select bitmap (by record) to a level bitmap (by column value), you only need to:

1. Generate a record bitmap using repetition level == 0; (Equal operator)
2. Use the record bitmap as a mask;
3. Call the extend operator to extend each bit of the select bitmap to the number of column values corresponding to each record;

- The resulting level bitmap can be used to filter field values in subsequent select operations.

Level Bitmap to Value Bitmap

```
7:  b_value := b_level
8:  selected_defs := ∅
9:  if defs ≠ ∅ then
10:     b_valid := equal(defs, max_def_level)
11:     b_value := compress(b_level, b_valid)
12:     selected_defs := select(defs, b_level)
13: selected_values := select(values, b_value)
14: return ⟨selected_reps, selected_defs, selected_values⟩
```



Goal:

Extract the positions corresponding to non-null values from the level bitmap and obtain the value bitmap, which is used to select the actual field values.

Compress operator

- Scan valid bitmap and find every position that is 1

- Then extract the bits at these positions from level bitmap and arrange them in order to form a new value bitmap

$$b_{value} = \text{PEXT}(b_{level}, b_{valid})$$

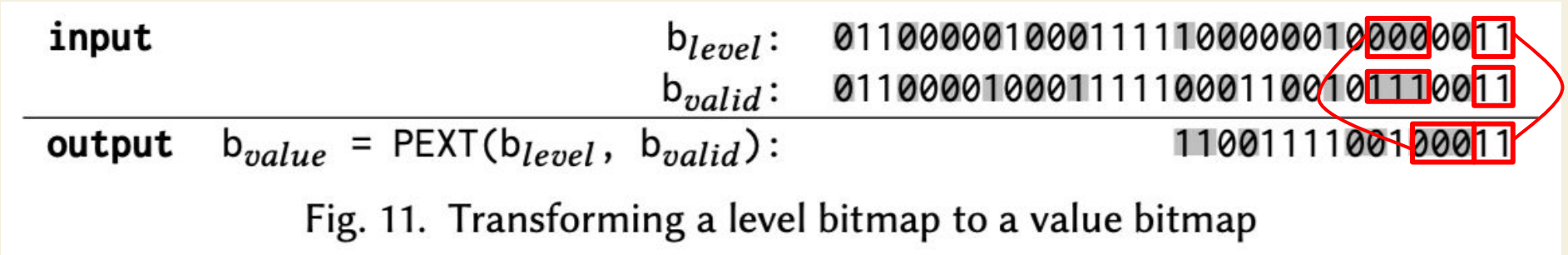
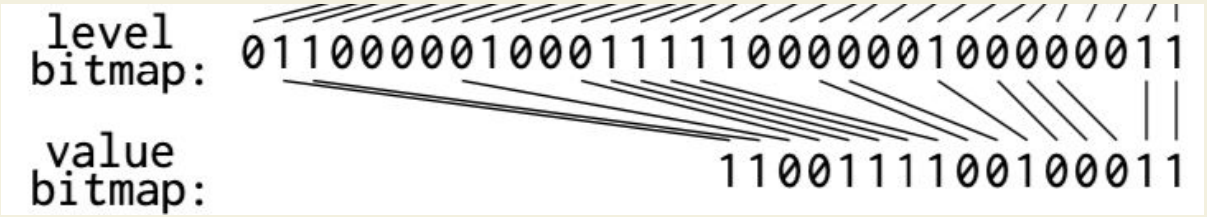


Fig. 11. Transforming a level bitmap to a value bitmap

Level Bitmap to Value Bitmap

```
7:  b_value := b_level
8:  selected_defs := ∅
9:  if defs ≠ ∅ then
10:     b_valid := equal(defs, max_def_level)
11:     b_value := compress(b_level, b_valid)
12:     selected_defs := select(defs, b_level)
13: selected_values := select(values, b_value)
14: return ⟨selected_reps, selected_defs, selected_values⟩
```



1. Generates a valid bitmap where each bit is 1 if the value is non-null, and 0 if it's null, by comparing each definition level to the maximum definition level.

2. Use the valid bitmap as a mask

3. Call the compress operator to extract the valid bits from the level bitmap, to form a new value bitmap

Workflow

- It is worth pointing out that, all operators used in Algorithm are bit-parallel algorithms.

- Additionally, all operators rely on either the PDEP or PEXT instruction to achieve the full data parallelism available in processor words.

Algorithm 4 select-parquet ($reps$, $defs$, $values$, b_{select})

1: $b_{level} := b_{select}$
2: $selected\_reps := \emptyset$
3: **if** $reps \neq \emptyset$ **then**
4: $\quad b_{record} := \text{equal}(reps, 0)$
5: $\quad b_{level} := \text{extend}(b_{select}, b_{record})$
6: $\quad selected\_reps := \text{select}(reps, b_{level})$
7: $b_{value} := b_{level}$
8: $selected\_defs := \emptyset$
9: **if** $defs \neq \emptyset$ **then**
10: $\quad b_{valid} := \text{equal}(defs, max\_def\_level)$
11: $\quad b_{value} := \text{compress}(b_{level}, b_{valid})$
12: $\quad selected\_defs := \text{select}(defs, b_{level})$
13: $selected\_values := \text{select}(values, b_{value})$
14: **return** $\langle selected\_reps, selected\_defs, selected\_values \rangle$

1. Introduction & Background
2. Bit Parallel Select Operator
3. Selection Pushdown
4. Selection Pushdown in Parquet
5. **Evaluation & Conclusion**

Evaluating Selection Performance

Experiment Objective

- To analyze how performance is affected by two key factors:
  - The **bit width** of column values
  - The **selectivity** of the select bitmap

Experimental Results

- **Parquet-Select outperforms original Parquet in all cases**
- The **lower the selectivity**, the **greater the speedup**
  - Because Parquet-Select only decodes selected values, reducing decoding cost
- **Smaller bit widths** lead to greater performance gains for Parquet-Select
  - Select operator can process **more values per CPU word** → higher data parallelism

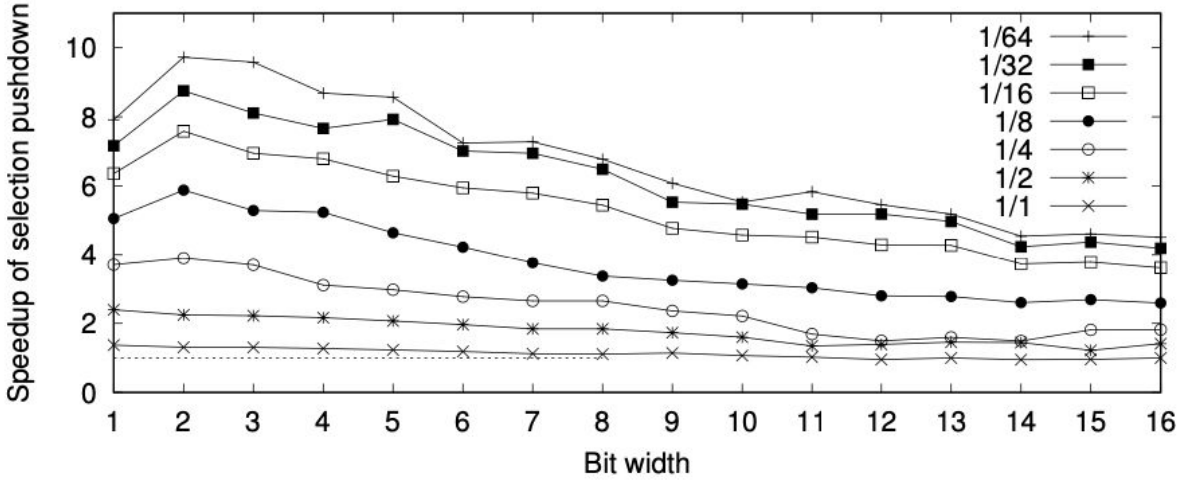


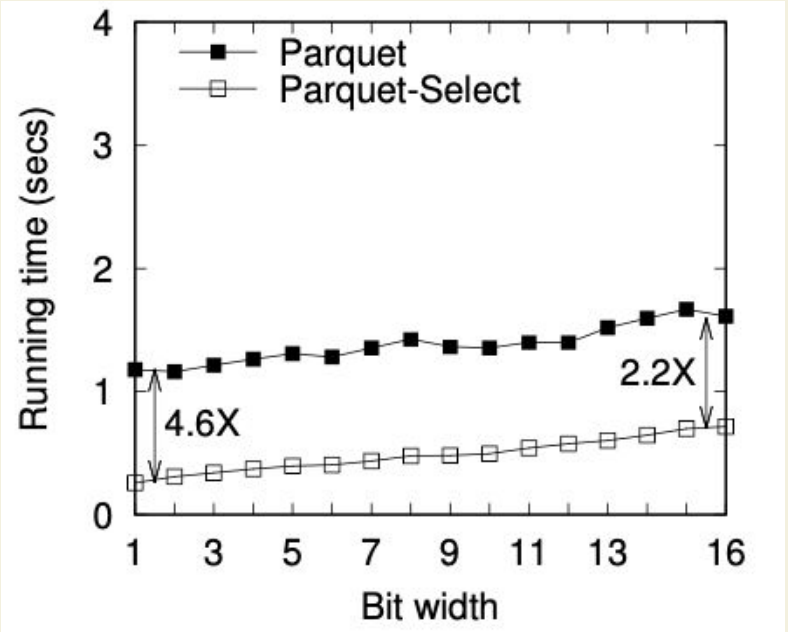
Fig. 12. Parquet vs. Parquet-Select: selection operation

Micro-Benchmark Evaluation

Use the following SQL query to test: SELECT MAX(a10), MAX(a11), ... , WHERE a1 < C1 AND a2 < C2 AND ...

Effect of Bit Width (a)

- **Parquet-Select performs better** with **smaller bit widths**.
- However, the **absolute execution time remains nearly the same**.
- **Unpack is the performance bottleneck**, and is **independent of bit width**.

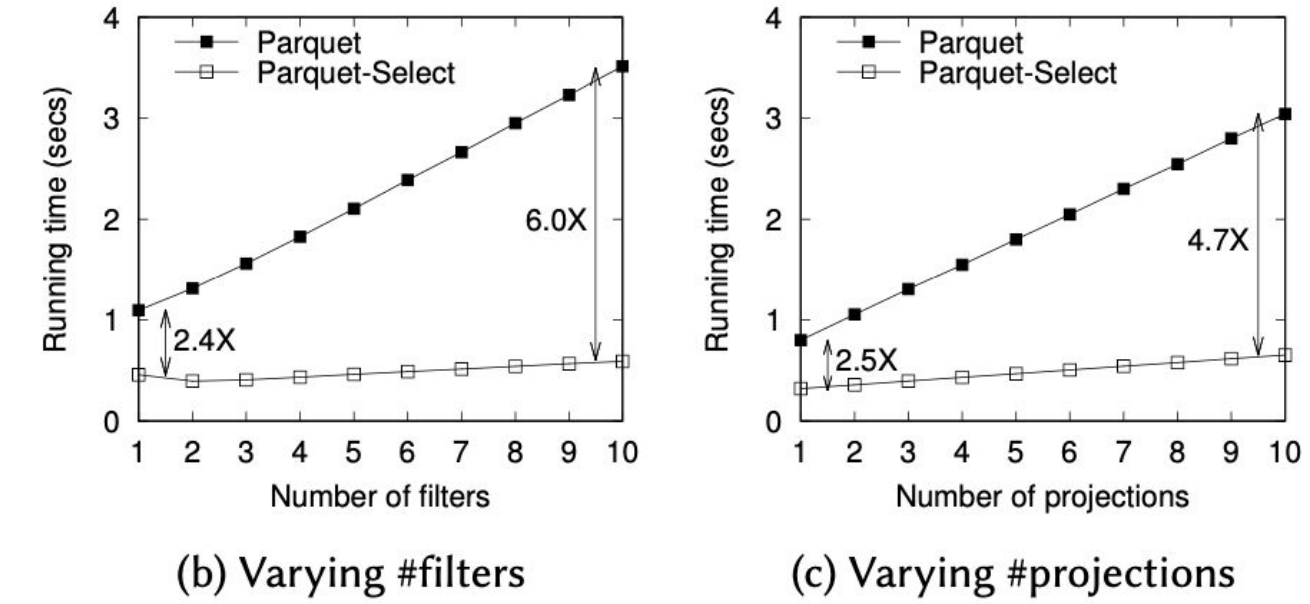


(a) Varying bit width

Micro-Benchmark Evaluation

Effect of Increasing Filters or Projections (b,c)

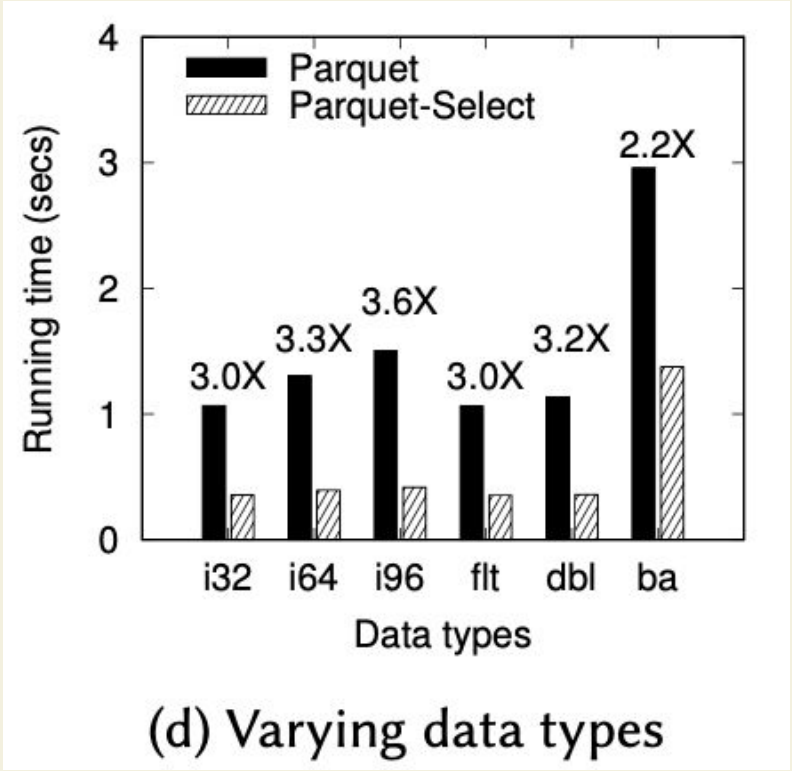
- The **more filters or projection columns**, the **better Parquet-Select performs**.
  - Multiple filters executed sequentially **reduce selectivity step by step**.
  - Later projection **process fewer values**, leading to more gains.



Micro-Benchmark Evaluation

Performance Across Data Types (d)

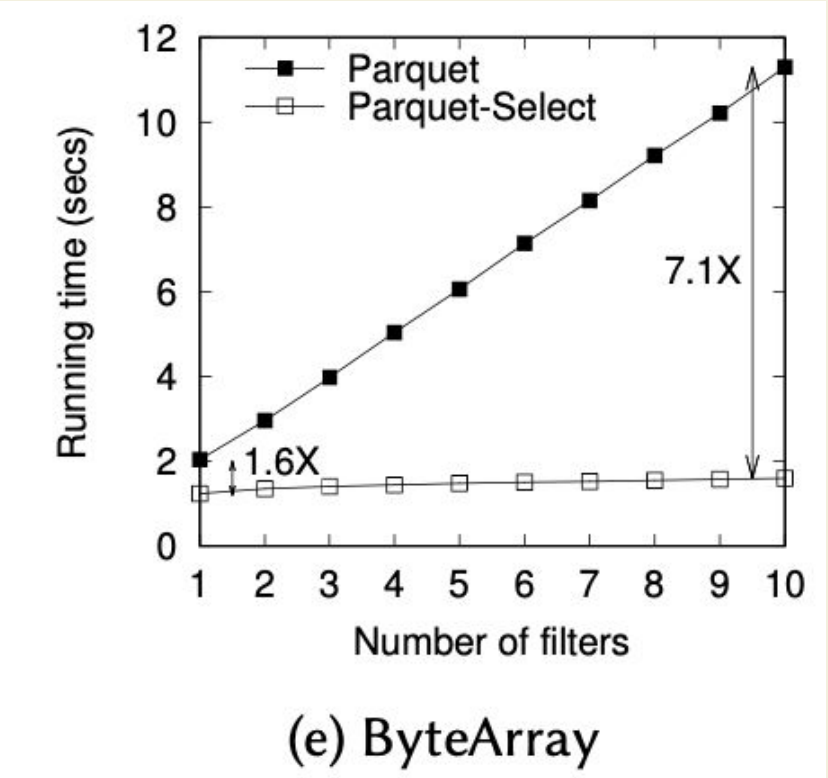
- For all types except byte array (e.g., **int32, int64, int96, float, double**), **Speedup ranges from 3.0× to 3.6×**
- **Byte array shows lower speedup** because:
  - The **first filter still needs to process all values**
  - **Predicates on byte arrays are more expensive**, which reduces early performance gain



Micro-Benchmark Evaluation

Effect of Filters for Byte Arrays (e)

- As more filters are added, **selectivity drops**, and **Parquet-Select's advantage increases**
- This shows that even for **expensive data types**, having enough filtering **can still bring significant speedups**



Case Study: TPC-H Benchmark Q6 (More realistic workload)

Non-nullable Columns

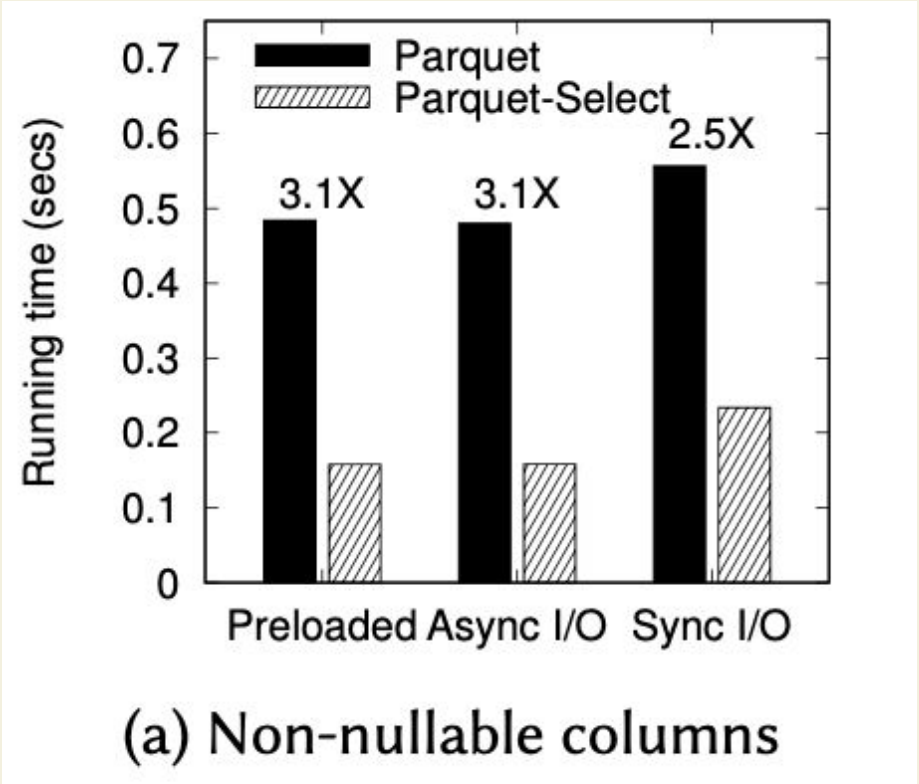
- All fields in the original dataset are **non-null**.
- Compared the query time under **three I/O modes**:

  preloaded

  asynchronous I/O

  synchronous I/O

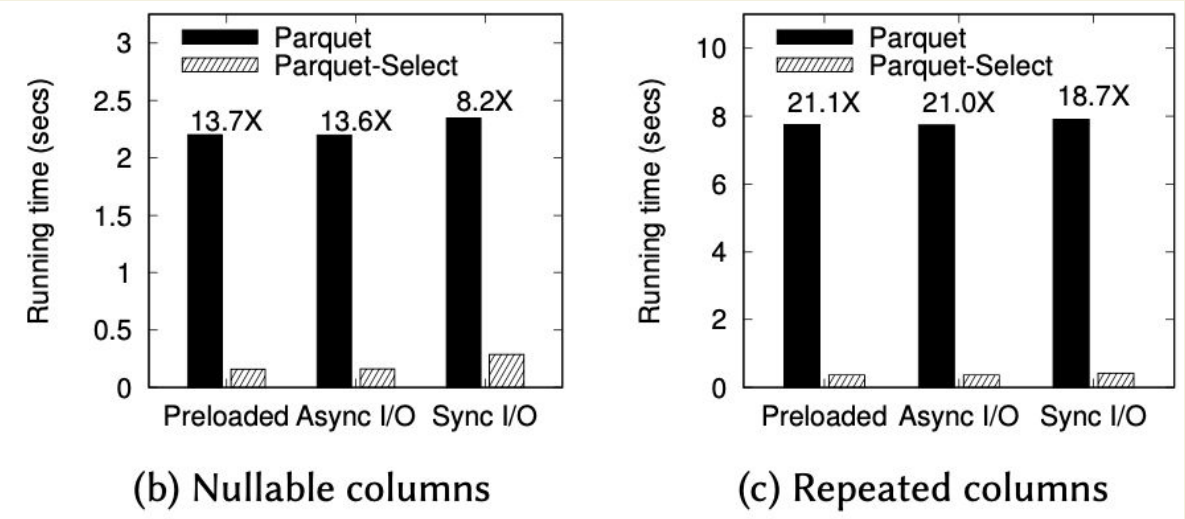
- lower speedup in synchronous I/O reflects the **limitations of CPU-side optimizations when I/O dominates the workload.**



Case Study: TPC-H Benchmark Q6 (More realistic workload)

Nullable and Repeated Columns

- Original Parquet query time **increased,** since parquet must **decode definition/repetition levels** and check for nulls/repeated values
- **Parquet-Select**, by contrast:
  - Evaluates **directly on encoded levels**
  - Uses **BMI instructions** to accelerate bitmap transformation



TPC-H Benchmark Evaluation

Experiment Objective

- To **integrate Parquet-Select into Apache Spark** and evaluate its performance under a real-world big data query engine.
- To compare two approaches:

  Spark + Parquet and Spark + Parquet-Select

Query Selection & Selectivity

- Evaluated **10 queries** from the **TPC-H benchmark**, with varying selectivity levels.

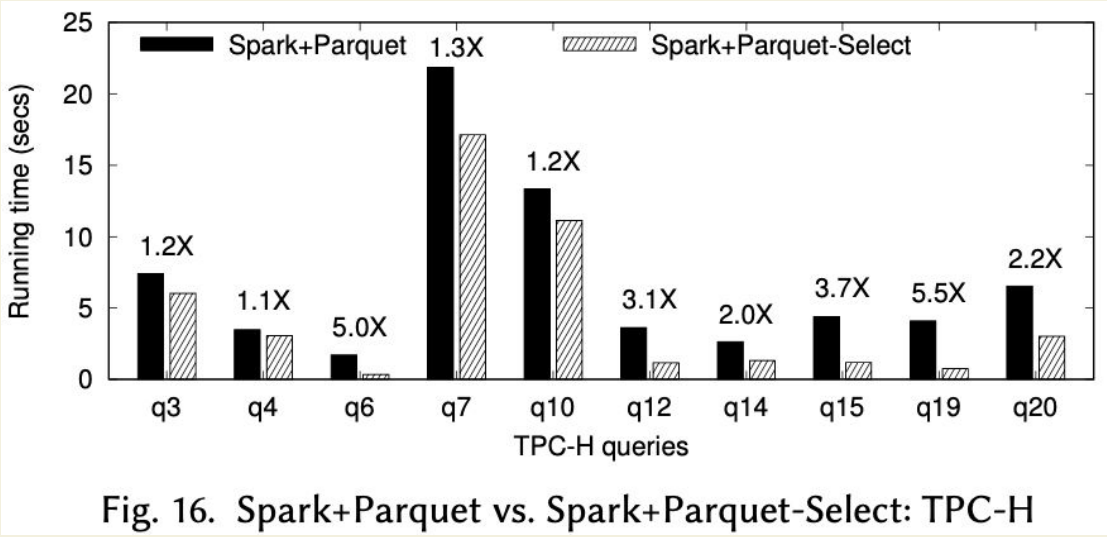


Fig. 16. Spark+Parquet vs. Spark+Parquet-Select: TPC-H

Q3	Q4	Q6	Q7	Q10	Q12	Q14	Q15	Q19	Q20
54%	63%	1.9%	30%	25%	0.5%	1.2%	3.8%	2.1%	15%

TPC-H Benchmark Evaluation

Results

- **Parquet-Select outperforms Parquet** across all queries.
- Speedup ranges from **1.1× to 5.5×**.
- The **lower the selectivity**, the **greater the performance gain**.

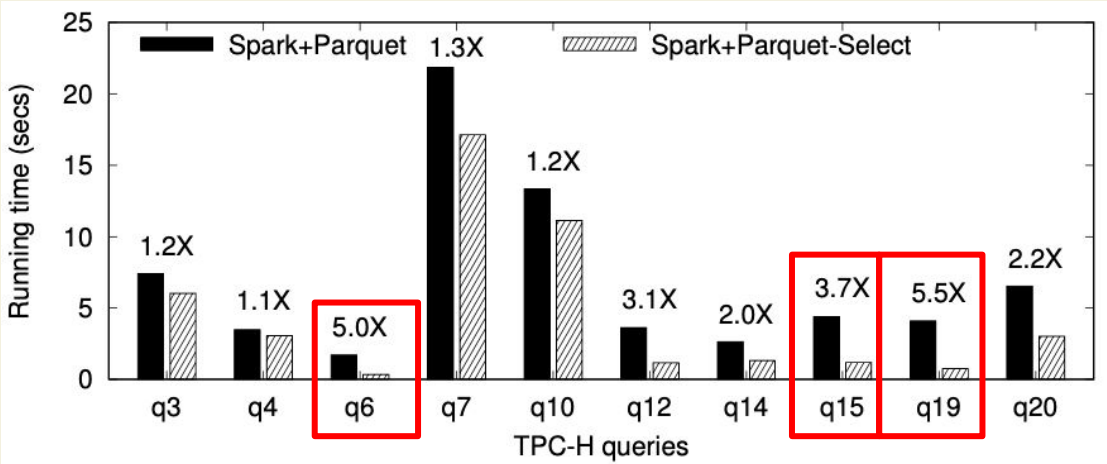


Fig. 16. Spark+Parquet vs. Spark+Parquet-Select: TPC-H

Reason

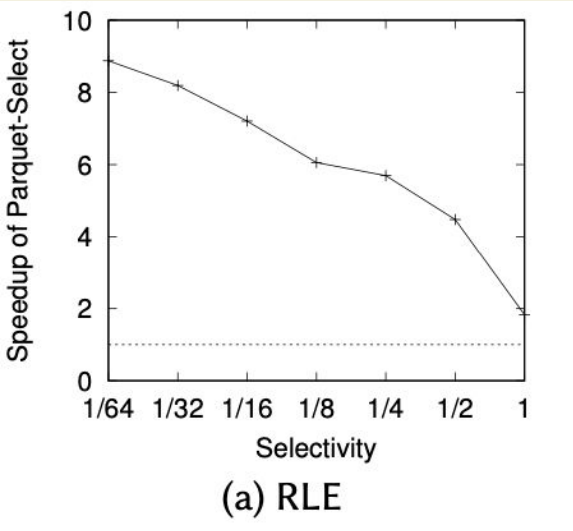
Parquet-Select significantly reduces the amount of data that needs to be **read and decoded**.

Q3	Q4	Q6	Q7	Q10	Q12	Q14	Q15	Q19	Q20
54%	63%	1.9%	30%	25%	0.5%	1.2%	3.8%	2.1%	15%

Impact of Encoding and Compression

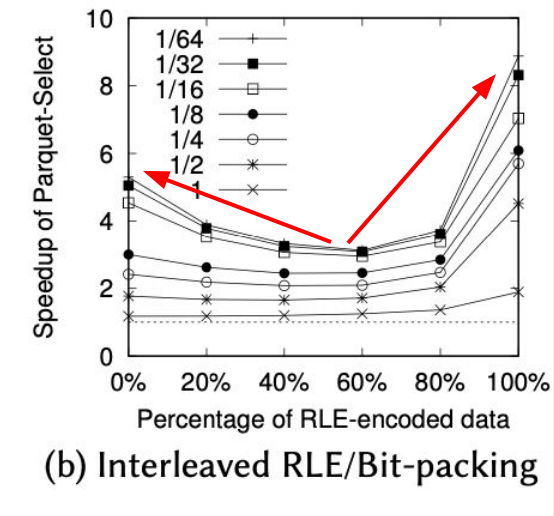
RLE Encoding (Run-Length Encoding)

- Even when selectivity = 1, Parquet-Select is still **1.8× faster**
  - Because Parquet-Select performs **decoding and filtering in a single pass**
  - Parquet requires **two passes**: one for decoding, one for filtering



Mixed Encoding

- Simulated with **interleaved RLE and bit-packing runs**
- **Best performance** is achieved when the column is encoded **purely as RLE or bit-packing**



Impact of Encoding and Compression

Page-Level Compression

- LZ4/Snappy reduced file size by ~30%
- **Both Parquet and Parquet-Select experienced performance degradation**:
  - **Decompression overhead outweighs I/O savings**
- Since decompression cost affects both systems equally:
  - The **absolute performance advantage** of Parquet-Select remains

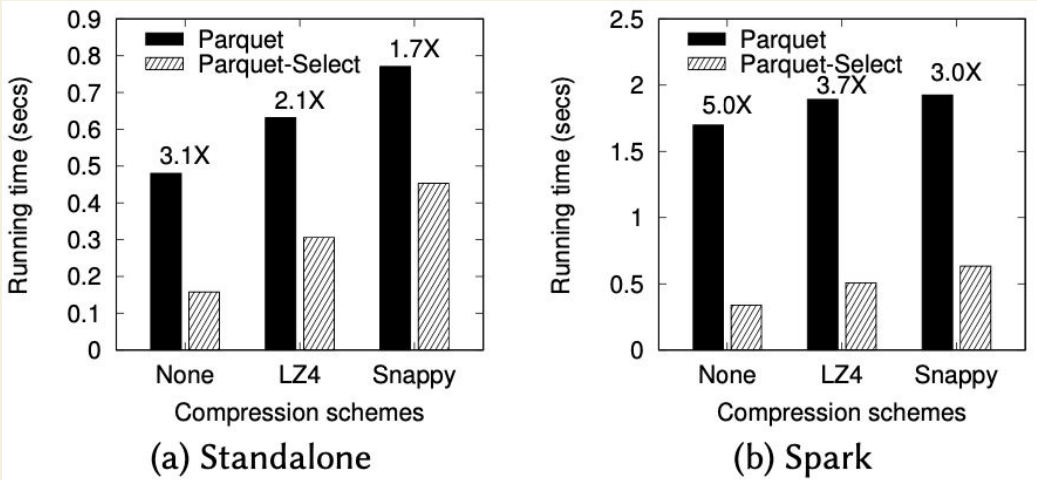


Fig. 18. Impact of page-level compression on TPC-H Q6

Discussion

Does the reliance on PEXT/PDEP instructions affect the portability of the proposed approach across different hardware platforms?

The Principles are Portable

- The Implementation described in the paper (using PEXT/PDEP) is architecture-specific.
- The high level idea described in the paper (leverage bit operations to compact data based on a bitmask) is architecture-independent.
- Creating an ARM based implementation could be a very interesting and logical future research direction.

ARM has its own bit instruction set (A64), as well as extensions like NEON and SVE2 that perform similar functions to Intel/AMD counterparts.

Could the overhead introduced by the select operator outweigh its benefits?

Select Operator Overhead - Response

- In Short: Not Really!
- All we're doing is changing the operation order
- The Difference is this way we're unpacking less data
- Speedup is negligible only in the worst case

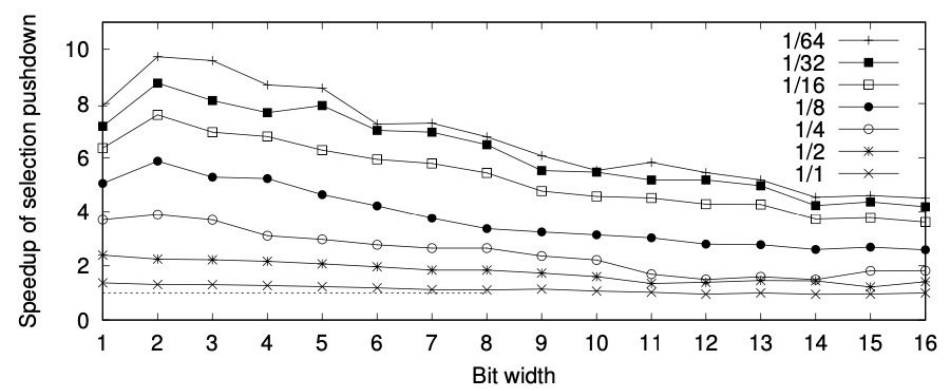
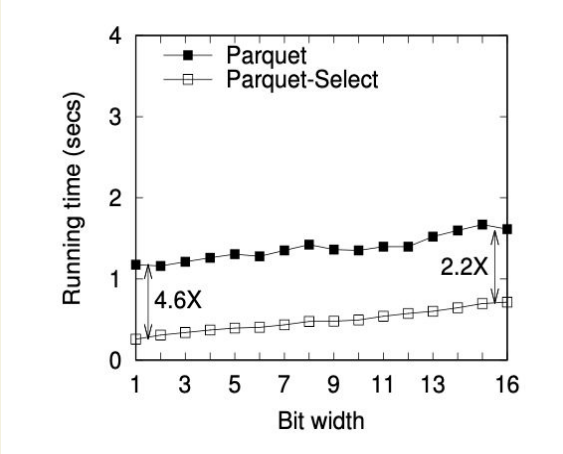


Fig. 12. Parquet vs. Parquet-Select: selection operation

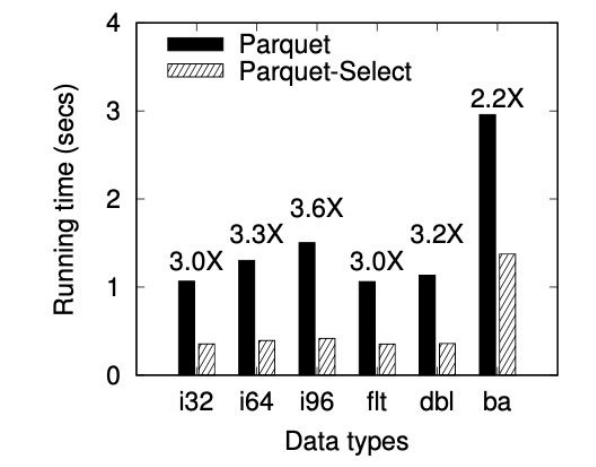
Is the select operator robust to varying bit-widths and value alignments?

Unpacking Cost and Bit Width are (largely) Independent!

It doesn't really matter what the width of the input values are since the size of the decoded values are fixed.



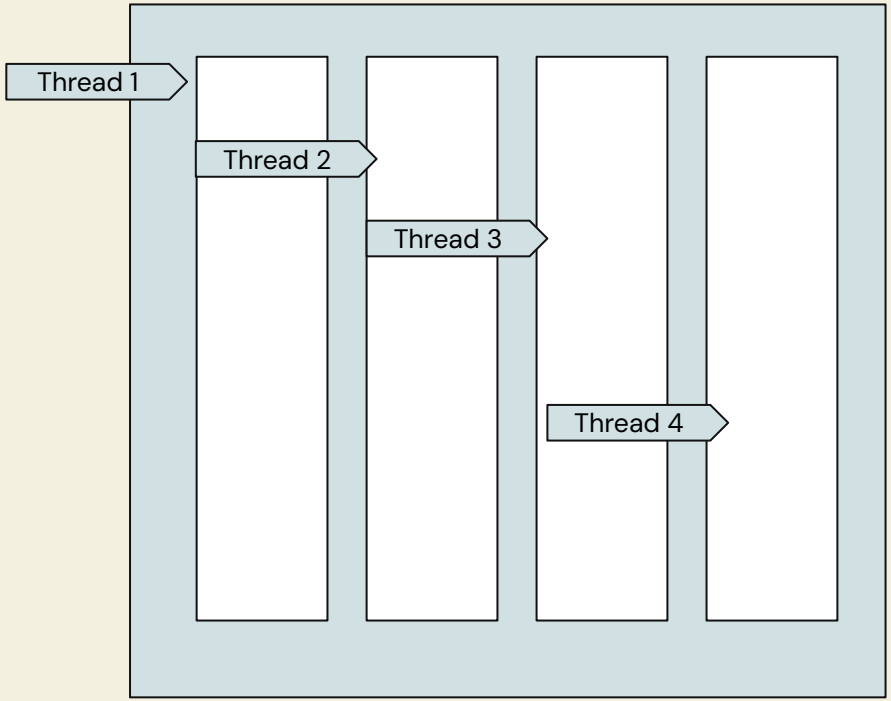
This extends to the different data types being operated on.



How would running experiments in a multi-threaded environment affect performance?

Multi-threaded Environment Performance - Response

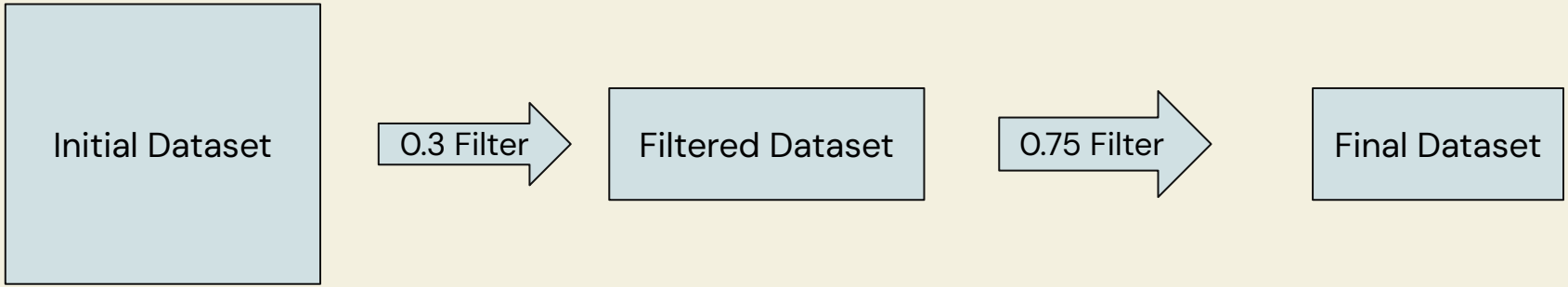
- Theoretically, should improve
- Bitmaps Are Naturally Parallelizable
- Multiple Threads can work on different segments of the bitmap concurrently.
- Multi-threaded experiments were not part of this paper, but it is another reasonable next step for research.



Are there any scenarios where the greedy algorithm for filter order would result in the wrong order, especially in real-world systems?

Filter Order - Response

- Greedy Algorithm used to select “optimal” ordering of filters
- Assumes Filters are all Independent and Selectivity is known in advance
- Choose starting filter: Optimal Filter Order -> Ascending Selectivity Order
  - Should minimize size of result for each subsequent filter
  - Lowest overall result is selected
- Does this always give the optimal ordering? – Only if the assumptions are true
  - Only checks O(n) of the O(n!) possible orderings
  - Remember the core idea of Query Optimization
- Relaxing Assumptions could lead to cool future work



How does the scalar BMI-Based approach interact with the SIMD-Based vectorized query engines?

BMI Complements SIMD Operations

- SIMD Operations are used to efficiently unpack and evaluate the values (Unpack/Evaluate)
  - Previous Limitation: decoded value sizes limit degree of parallelism
- Bit Manipulation Instructions reduce the total values that need to be unpacked. (Select/Transform)
  - Fewer values to unpack → Smaller total decoded value size

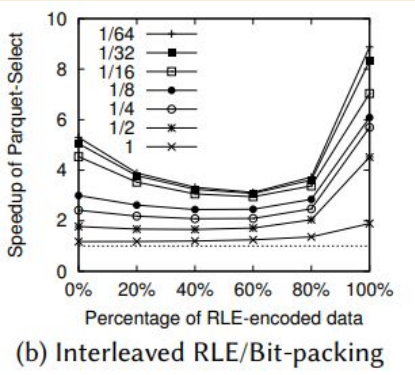
What performance bottlenecks are observed in the experiments?

Initial Query

- When a query starts with no prior filter, Parquet-Select must unpack all the data to create the bitmap for the next operation

Switching Between RLE and Bit-Packing

- Parquet-Select handles both run-length encoding (RLE) and bit-packing, but flipping between them adds overhead



(b) Interleaved RLE/Bit-packing

Parallel predicate evaluation assumes order-preserving encodings and simple predicates on encoded data. Should these methods be extended to support more complex filters?

After optimizing unpacking, what other parts of the selection pushdown pipeline offer the most potential for further improvement?

Evaluation: Improving Predicate Pushdown

- Predicate pushdown techniques are currently limited to order-preserving encodings and simple predicates, which don't have significant real world application
- However, developing techniques to encode and evaluate more complex predicates directly on encoded values could eliminate/reduce the need for unpacking in certain operations
- Can help resolve initial query bottleneck

Parquet-Select shows impressive performance gains, but are there specific benchmarks or workloads where these improvements diminish or break down?

High Selectivity Queries

- Low selectivity means fewer values to unpack down the line
- Figure 12 shows speedup drops as selectivity rises
- As selectivity goes to 100%, Parquet-Select's performance essentially converges with Parquet
- Why? If we're unpacking everything anyway, the BMI select step adds overhead without saving much time

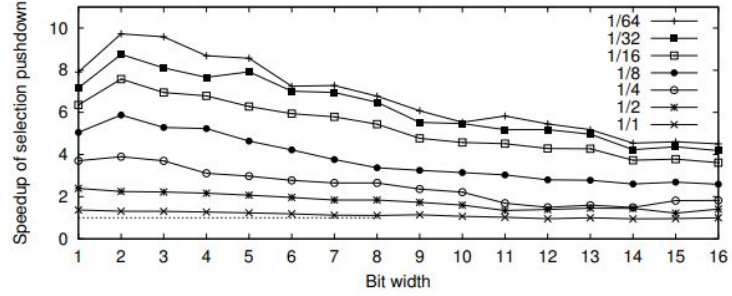


Fig. 12. Parquet vs. Parquet-Select: selection operation