Selection Pushdown in Column Stores using Bit Manipulation Instructions

Rohit Vemparala - U50993392

Speed-up query execution and improve performance in columnar data stores - propose techniques and empirically evaluate using micro-benchmarks and TPC-H benchmark

COLUMN STORES

Data is organized and stored in columns - values in each column are stored contiguously

In contrast to row-oriented stores which store data in a row contiguously

Name	Age	City
John Smith	30	New York
Jane Doe	25	Chicago
Bob Johnson	35	Miami

ROW DATA-STORE

Row1: [“John Smith”, 30, “New York”]

Row2: ["Jane Doe", 25, "Chicago"]

Row3: [“Bob Johnson”, 35, “Miami”]

COLUMN DATA-STORE

Name: [“John Smith”, “Jane Doe”, “Bob Johnson”]

Age: [30, 25, 35]

City: [“New York”, “Chicago”, “Miami”]

Why use column stores?

Data compression - Store data more compactly due to similar/repeating values in columns - aggressive encoding schemes - **dictionary encoding**

Query performance - Only read/process necessary columns - reduces I/O, improves performance

Scalability - Easily add new columns to existing data, individually compress, index each column - better storage utilization

OPTIMIZATIONS

Data Compression

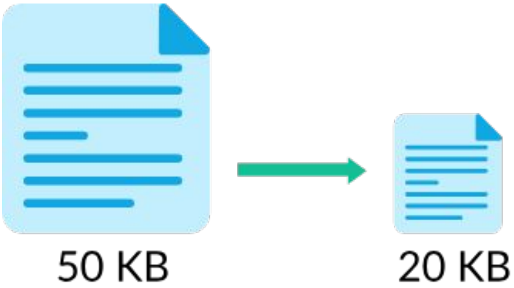
Use fewer bits than original representation to store data.

Encoding Scheme - A way to store values efficiently, in-order to compress them

Popular scheme - **dictionary encoding**

Dictionary Encoding - Each distinct value in column mapped to a unique **short-code**

Usually store these in a **bit-packed manner**



Dictionary Encoding

A loss-less compression technique that stores each unique value of a column in memory and associate each record with its corresponding unique value.

Example

To the swinging and the ringing of the bells,

bells, bells-of the bells, bells, bells, bells

Bells, bells, bells- To the rhyming and the

chiming of the bells!

Word	Reference	Binary
To	0	0000
the	1	0001
swinging	2	0010
and	3	0011
ringing	4	0100
of	5	0101
bells	6	0110
Bells	7	0111
rhyming	8	1000
chiming	9	1001
,	10	1010
-	11	1011
!	12	1100

Original

To the swinging and the ringing of the bells, bells, bells-of the bells, bells, bells, bells Bells, bells, bells- To the rhyming and the chiming of the bells!

Encoded

0 1 2 3 1 4 5 1

6 10 6 10 6 11 5 1 6 10 6 10

6 10 6 10 7 10 6 10 6 11 0 1

8 3 1 9 1 3 5 1 6 12

Bit-packing

Smart way of storing data - make your data-representation fit your data

Example

Boolean array

00000001	00000000	00000001
TRUE	FALSE	TRUE

→

Bit array/Bitmask/Bit map/Bit set

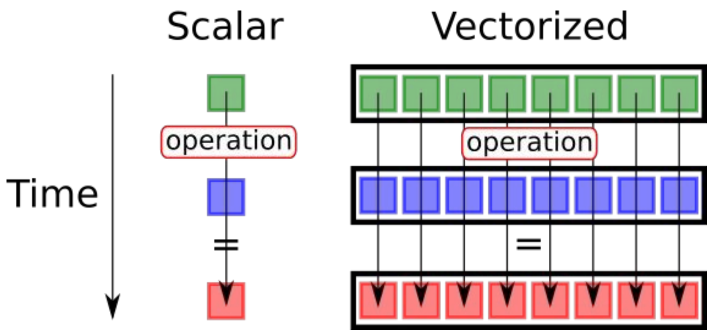
00000101		
TRUE	FALSE	TRUE

Bit Packing + Dictionary Encoding

- **Reduces storage cost**
- **Increased query processing latency due to decoding**

Making this process more efficient - **SIMD Vectorization** (Single Instruction/Multiple Data)

Accelerating Decoding Process - SIMD vectorization



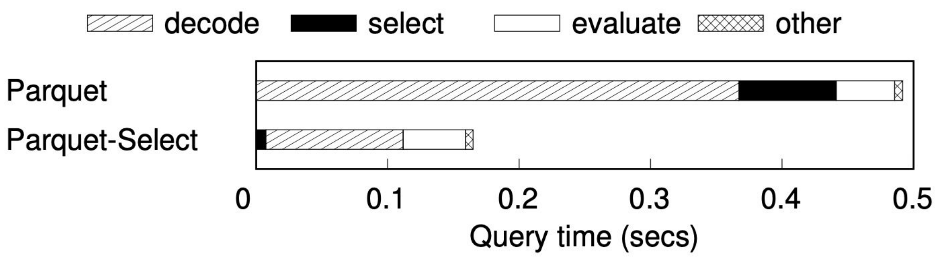


Fig. 1. Time breakdown of TPC-H Q6

Parquet adopts this method

Still decoding takes up majority of query execution time

Fundamental limitation - Produced decoded value much bigger than encoded value (obviously) - limiting degree of **data parallelism**

To alleviate this - avoid decoding all together - **predicate pushdown**

Predicate Pushdown

A query optimization technique to filter data at the data source

Predicates - Conditions or filters

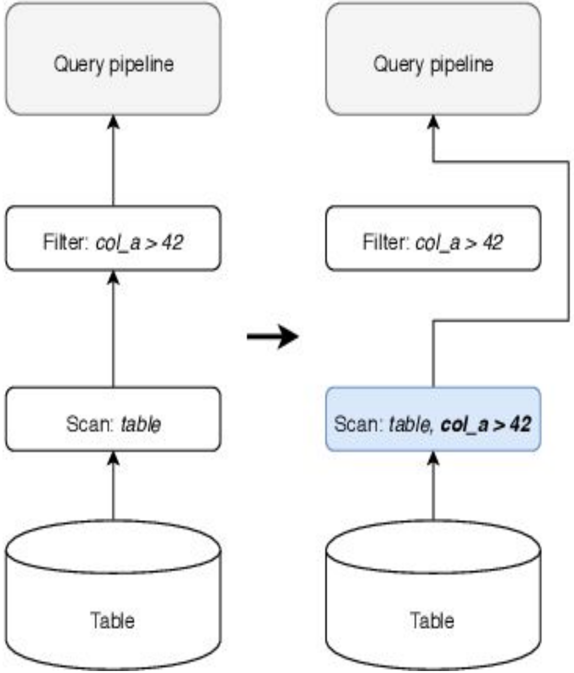
eg. **WHERE** clause

Benefits

- Reduced Data Transmission
- Decreased Processing Times
- Enhanced Performance

```sql
SELECT [AddressID]
      ,[AddressLine1]
      ,[AddressLine2]
      ,[City]
      ,[StateProvinceID]
      ,[PostalCode]
      ,[SpatialLocation]
      ,[rowguid]
      ,[ModifiedDate]
  FROM [AdventureWorks2016CTP3].[Person].[Address]
  WHERE City='Seattle'
  AND AddressLine1 like '%Bradford%'
```

Filter Predicates



Typical execution

1. **Query** — Query with predicates sent to database engine
2. **Retrieval** — Database engine queries the datasource which returns all data
3. **Filtering** — Filters applied to discard unnecessary rows
4. **Result** — Filtered rows returned as result for query

Predicate Pushdown

1. **Query** — Query with predicates sent to database engine
2. **Pushdown** — Database engine pushes down predicates to datasource
3. **Filtering** — Data evaluated with predicates and only relevant data fetched
4. **Result** — Filtered rows returned as result for query

Predicate Pushdown + Encoding

Evaluate converted predicate on encoded value directly - pushing down predicate evaluation - avoid costly decoding

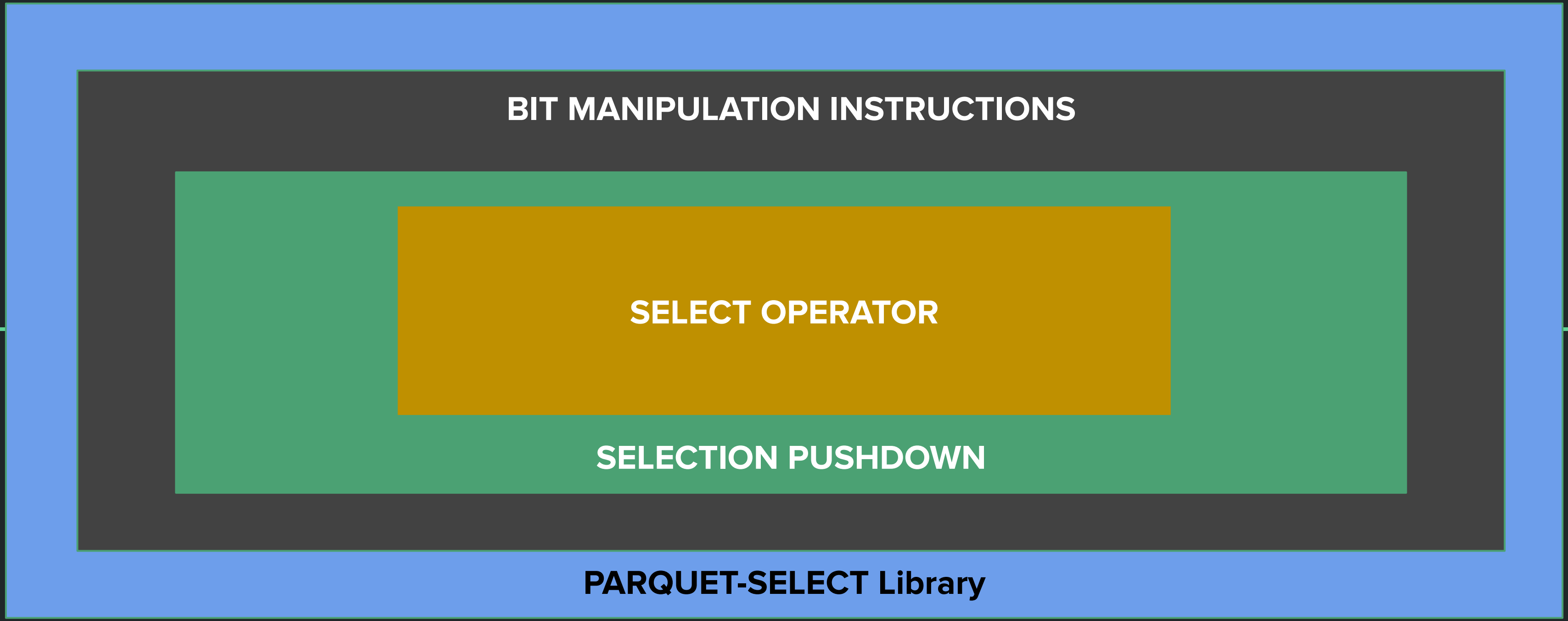
Even if high performance - still rely on 2 key assumptions -

1. Encoding is order preserving
2. Simple predicates (i,e basic comparison) - can be converted to equivalent encoded form

Neither hold true in practice - eg. Parquet uses dictionary encoding - not order-preserving - eliminates possibility of using this technique in Parquet

Also - complex predicates - string matching, user-defined functions, cross-table predicates - not supported

SOLUTION OVERVIEW



BIT MANIPULATION INSTRUCTION SET

Extension for X86 architecture for Intel, AMD

Improve the speed of bit manipulation and common bitwise operators using dedicated hardware instructions. They are **non-SIMD** and operate only on general-purpose registers.

Even before BMI, bitwise operations were used in database applications - implemented in software

Software implementations can be **replaced** with BMI without reworking algorithms - take advantage of speedup provided by hardware implementation

Encoding	Instruction	Description	Equivalent C expression
VEX.LZ.0F38 F2 /r	ANDN	Logical and not	`~x & y`
VEX.LZ.0F38 F7 /r	BEXTR	Bit field extract (with register)	`(src >> start) & ((1 << len) - 1)`
VEX.LZ.0F38 F3 /3	**BLSI**	**Extract lowest set isolated bit**	**`x & -x`**
VEX.LZ.0F38 F3 /2	BLSMSK	Get mask up to lowest set bit	`x ^ (x - 1)`
VEX.LZ.0F38 F3 /1	BLSR	Reset lowest set bit	`x & (x - 1)`

F3 0F BC /r	TZCNT	Count the number of trailing zero bits	`31 + (!x)`
			`- (((x & -x) & 0x0000FFFF) ? 16 : 0)`
			`- (((x & -x) & 0x00FF00FF) ? 8 : 0)`
			`- (((x & -x) & 0x0F0F0F0F) ? 4 : 0)`
			`- (((x & -x) & 0x33333333) ? 2 : 0)`
			`- (((x & -x) & 0x55555555) ? 1 : 0)`

Encoding	Instruction	Description
VEX.LZ.0F38 F5 /r	BZHI	Zero high bits starting with specified bit position [src & (1 << inx)-1];
VEX.LZ.F2.0F38 F6 /r	MULX	Unsigned multiply without affecting flags, and arbitrary destination registers
VEX.LZ.F2.0F38 F5 /r	**PDEP**	**Parallel bits deposit**
VEX.LZ.F3.0F38 F5 /r	**PEXT**	**Parallel bits extract**
VEX.LZ.F2.0F3A F0 /r ib	RORX	Rotate right logical without affecting flags
VEX.LZ.F3.0F38 F7 /r	SARX	Shift arithmetic right without affecting flags
VEX.LZ.F2.0F38 F7 /r	SHRX	Shift logical right without affecting flags
VEX.LZ.66.0F38 F7 /r	SHLX	Shift logical left without affecting flags

(a) PEXT (parallel bit extract) (b) PDEP (parallel bit deposit)

Fig. 2. Examples of PEXT and PDEP

PEXT

Extracts the bits selected by a select mask operand from a source operand and copies them to the contiguous low-order bits in the destination, with the high-order bits set to 0s.

PDEP

Opposite of PEXT: the contiguous low-order bits from the source operand are copied to the selected bits of destination, indicated by the select mask operand, while other bits in the destination are set to 0s.

```rust
fn pext64(word: u64, mask: u64) -> u64 {
    let mut out = 0;
    let mut out_idx = 0;

    for i in 0..64 {
        let ith_mask_bit = (mask >> i) & 1;
        let ith_word_bit = (word >> i) & 1;
        if ith_mask_bit == 1 {
            out |= ith_word_bit << out_idx;
            out_idx += 1;
        }
    }

    out
}
```

```rust
fn pdep64(word: u64, mask: u64) -> u64 {
    let mut out = 0;
    let mut input_idx = 0;

    for i in 0..64 {
        let ith_mask_bit = (mask >> i) & 1;
        if ith_mask_bit == 1 {
            let next_word_bit = (word >> input_idx) & 1;
            out |= next_word_bit << i;
            input_idx += 1;
        }
    }

    out
}
```

BITSTRING	a	b	c	d	e	f	g	h
MASK	**1**	0	**1**	0	0	**1**	**1**	0
PEXT	0	0	0	0	**a**	**c**	**f**	**g**
PDEP	**e**	0	**f**	0	0	**g**	**h**	0

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

BMI **2 orders** of magnitude faster than author's software implementation

BLSI - Software implementation much faster than BMI!

PEXT, PDEP - BMI much faster than software implementation

SIMD vs BMI

Feature	SIMD (Single Instruction, Multiple Data)	BMI (Bit Manipulation Instructions)
Purpose	Enables parallel processing of multiple data elements	Provides operations for low-level bit manipulation
Operations	Operates on vectors of data elements simultaneously	Operates on individual or groups of bits within data
Use Cases	Multimedia processing, scientific computing, etc.	Cryptography, compression algorithms, hashing, etc.
Examples	Intel's SSE, AVX, ARM's NEON	Intel's BMI1, BMI2
Performance Impact	Improves performance by parallelizing operations	Provides efficient bit-level manipulation
Granularity	Operates on data elements (e.g., floats, integers)	Operates on individual bits or groups of bits

APACHE PARQUET

Open-source columnar storage format - initiated by twitter and cloudera, inspired by Dremel

The parquet format is -

- Supports nested columns
- Binary format
- Encoded
- Compressed
- Storage efficient

Sample Data

Name	Phone	Address
John	1001	{road: road1, street: street1}
Sten	1002	{road: road1, street: street1}
Jakob	1003	{road: road1, street: street1}
Milne	1004	{road: road1, street: street1}

Column Name	Address
Optional/Required/Repeated	Optional
Data Type	Binary
Encoding Info for Binary	0:UTF8
Repetition value	R:0
Definition value	D:0

FAST SELECT OPERATOR

Used for selecting specific values , output them contiguously and remove unselected values


```
                           v7  v6  v5  v4  v3  v2  v1  v0
input              values: 0101101110110100000100111 1001010
            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): 00010000000000000000000100000000
     extended = high - low: 11111111000000000000111100000000
      step 2: select values based on the extended bitmap
output                                             v7  v6  v2
    PEXT(values, extended):                    010110110011
```

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

Required Output - v7, v6, v2

Naive solution - Go over each bit-packed value, extract ones needed - O(n) steps

Issue - Values much smaller than available word size (64bits) - inefficient utilization

Goal

More efficient select operation - something **bit-parallel** - simultaneously process all values packed into 64-bit processor word parallely

BIT PARALLEL ALGORITHM

Formal Definition - . For a given word size w, an algorithm is a bit-parallel algorithm if it processes n k-bit values in $O(nk/w)$ instructions.

Two cases -

Case 1: Word size is multiple of bit width - **Simplified Algorithm (3.2)**

Case 2: Word size is not a multiple of bit width - **General Algorithm (3.3)**

Simplified Algorithm

Bit width (k) power of 2 ∋ no value placed across word boundaries

1 bit values - Extract all bits that correspond to 1s in bitmap from values - exactly what **PEXT** does!

BITSTRING	a	b	c	d	e	f	g	h
MASK	**1**	0	**1**	0	0	**1**	**1**	0
PEXT	0	0	0	0	**a**	**c**	**f**	**g**

k bit values - Slightly general case - slight modification - instead of using select bitmap directly - need **extended bitmap** which has k bits - duplicate each bit in select bitmap k times

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*

Select **3** 4-bit values from **8** 4-bit values

Step 1 - Input select map to extended bitmap - the authors use only 3 instructions - 2 **PDEP**, one subtraction along with a mask from $0^{(k-1)}1$ where k power of 2

Step 2 - Use **PEXT** to get relevant values (v7, v6, v2)

```
                                    v7  v6  v5  v4  v3  v2  v1  v0
input                      values:  01011011101101000001001111001010
                    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
     step 2: select values based on the extended bitmap
output                                                  v7  v6  v2
     PEXT(values, extended):                            010110110011
```

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

Input bitmap	1	1	0	0	0	1	0	0
Extended bitmap	1111	1111	0000	0000	0000	1111	0000	0000

General Algorithm

Extend simplified algorithm - supports arbitrary bit width k

Challenge - Values can span across word boundaries - deal with these with minimal overhead

Previously used **Algorithm 1** still valid - with slight modifications

No additional overhead when compared to simplified algorithm!

Algorithm 3 generate_masks (w, k)

1: $masks := \varnothing$
2: **for** $i := 0$ **to** k **do**
3: $\quad offset := k - (i \times w) \ \% \ k$
4: $\quad masks.\text{add}((0^{k-1}1...0^{k-1}1 \ll offset) \vee 1)$
5: **return** $masks$

Algorithm is **bit-parallel** - run constant number of instructions on each processor word

```
                                     word 2                       (word 1 & 0)
                            v31v30v29v28v27v26v25v24v23v22v2      1...
input             values:   11000101110110000000101110111101      ...
           select bitmap:     1  0  1  0  0  0  0  1  0  0 0      ...
constant            mask:   00100100100100100100100100100101      ...
          step 1: transform the select bitmaps to extended bitmaps
    low = PDEP(bitmap,mask):  00100000100000000000000100000000      ...
 high = PDEP(bitmap, mask-1): 00000100000000000000001000000000      ...
        extended = high-low:  11100011100000000000011100000000      ...
             step 2: select values based on the extended bitmaps
output                                              v31v29v24       ...
      PEXT(values, extended):                        110011011       ...

      (word 2)                word 1                           word 0
        ...v2  1v20v19v18v17v16v15v14v13v12v11v   10 v9 v8 v7 v6 v5 v4 v3 v2 v1 v0
  values: ...  11011100010000111111010010111000   11101000001110010101110010100011
  bitmap: ...  0  0  0  1  0  0  0  0  0  0  11    1  1  0  0  0  0  0  1  0  0  0
    mask: ...  10010010010010010010010010010011   01001001001001001001001001001001
          step 1: transform the select bitmaps to extended bitmaps
     low: ...  00000000010000000000000000000011   01001000000000000000001000000000
    high: ...  00000010000000000000000000010010   01000000000000000001000000000000
extended: ...  00000001110000000000000000001111   11111000000000000000111000000000
             step 2: select values based on the extended bitmaps
          ...                          v18v11v                          10 v9 v3
  output: ...                          0011000                          11101110
```

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

SELECTION PUSHDOWN

AIM - Accelerate arbitrary scans making the best use of select operator discussed previously!

Scan operator - Return value of projection columns in query (i.e. SELECT) from records that match filters (i.e. WHERE clause)

Key Insight - Predicate Pushdown + Encoding/Compression + Select operator

Pushdown select operator to select encoded values directly

Technical Challenge - Fast Select operator on encoded values + move selected values from encoded vector values simultaneously

Based on simple observation - query usually involves multiple predicates across multiple columns - while evaluating predicates bypass records that fail prior predicates - short-circuit

Seems obvious, why is it not used by others?

Previous work **intentionally** ignore fact that some values were potentially filtered by other predicates as the cost of select operator outweighs potential cost savings in predicate evaluation

However, due to **fast select operator** - viable to filter at predicate level upfront

Key points is a framework that takes advantage of BMI based select operator for projecting and filtering along with technical challenges associated with this approach - which can also be addressed with BMI

The Idea

Each filter operation - produces select bitmap - 1 bit/record to indicate if record satisfies all previous filters or not.

Select bitmap - fed into next **filter**/**projection** operation - decrease data at each step

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

1. Filter on a - no bitmap
2. Filter on b - $bitmap_a$
3. Projection on c - $bitmap_a$
4. Final result - $selected_c$

column a	$bitmap_a$ = filter(a, null, < 10)
	$bitmap_a$: 10100001000001000000111000001000
column b	$bitmap_b$ = filter(b, $bitmap_a$, < 4)
	$bitmap_b$: 00100001000001000000010000000000
column c	$selected_c$ = project(c, $bitmap_b$)
	$selected_c$: v29v24v18v10

Fig. 5. Operations in evaluating the example query

Steps

Select

Use fast select operator to remove irrelevant values from the target column. Pushing down the select operator results in a reduced number of values that need to be passed to the subsequent operators.

Unpack

Convert encoded values to native representation in primitive data types with **SIMD based** implementation. For project operations, we can now return the unpacked results and skip the remaining two operators/steps

Evaluate

For filter operations, evaluate all decoded values with the filter predicate - generate a bitmap to indicate whether each value satisfies the predicate (allows arbitrary predicates). Selected values are now in primitive data types - **SIMD vectorization**

Transform

Evaluate operator bitmap may not be directly used as a select bitmap for the next operation as it has as many bits as the selected records, rather than all records. The transform operator is designed to convert such a bitmap into an appropriate select bitmap that can be used for the subsequent operation(s) - efficient way with BMI

Filter Ordering

So far - assumed filters evaluated in same order as in the query

SELECT c FROM R **WHERE a < 10 AND b < 4** - First a then b

Query optimization - what order to consider filters in - consider both bit width and filter selectivity

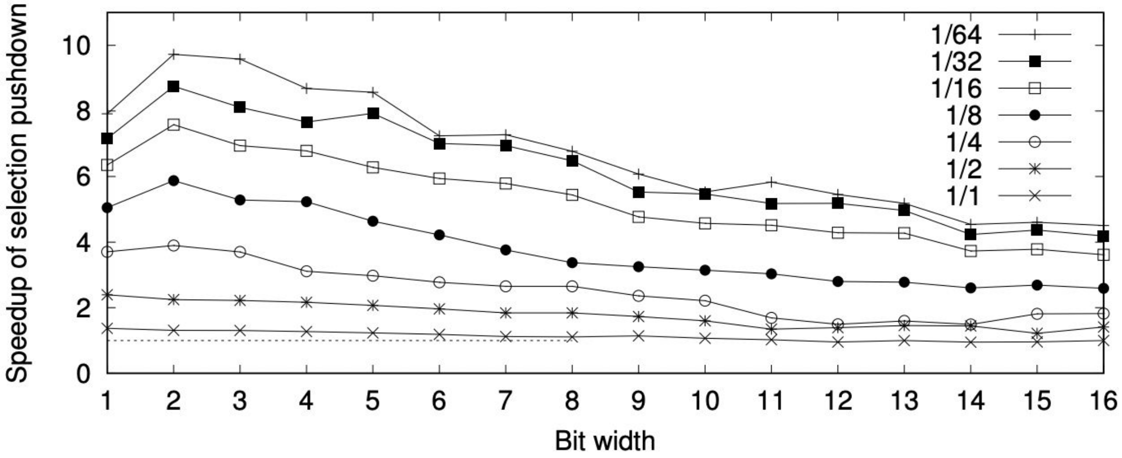


Fig. 12. Parquet vs. Parquet-Select: selection operation

Cost model and a **greedy model** to determine best order for filter evaluation

Assumptions: Filter predicates independent of each other, selectivity of each is known prior - via selectivity estimation techniques

Cost Model

$$cost(f_1) + \sum_{i=2}^{n} cost(f_i) \quad \propto \quad 1 + \sum_{i=2}^{n}(\frac{k_i}{w} + \prod_{j=1}^{i-1} s_j) \quad \propto \quad \sum_{i=2}^{n} \frac{k_i}{w} + \sum_{i=2}^{n} \prod_{j=1}^{i-1} s_j$$

k - bit width

w -processor word size

s - selectivity where $s \in [0, 1]$.

sequence of n filters, $f_1 \ldots f_n$.

Objective - Minimize cost of running filters

= cost(select) + cost(unpack+evaluate) + cost(transform) [Except first filter f_1]
$\propto k_i/w$

Bit Parallel Algorithm Formal Definition - For a given word size w, an algorithm is a bit-parallel algorithm if it processes n k-bit values in $O(nk/w)$ instructions.

Unpack + Evaluate - Run on subset of values selected by filter - number = total values * product of selectivity of previous filters

Transform - Ignore as only one **PDEP** instruction

First filter - No select, just unpack and evaluate

Minimize Cost

Observations

1. For sequences starting with the same filter, the term $\sum_{i=2}^{n} \frac{k_i}{w}$ remains unchanged and does not impact the overall cost, regardless of the order of the rest of the filters
2. To minimize the second term $\sum_{i=2}^{n} \prod_{j=1}^{i-1} s_j$, we should sort all filters in ascending order of selectivity, assuming the first filter has been determined.

It is evident that a **simple greedy approach** can find the optimal order

1. Select an arbitrary filter as the first filter
2. Optimal order of the remaining filters can be found by sorting them in the ascending order of their selectivity, whose cost can be calculated by using Equation 1
3. Compare all n possible choices for the first filter and find the one with the lowest overall cost.

This approach drastically reduces our search space from $O(n!)$ to $O(n)$ candidate sequences, and the obtained order is optimal under the aforementioned assumptions. Relaxing these assumptions is an interesting direction for future work

Extending Filter predicate assumption

Previously, filter predicate (**WHERE** clause) was assumed to be a conjunction of filters. It can be extended to other scenarios like conjunctions, disjunctions, negations, or an arbitrary boolean combination of them.

DISJUNCTION - Convert to a combination of conjunctions and negations by applying De Morgan's laws: $a \lor b = \neg(\neg a \land \neg b)$.

*SELECT c FROM R WHERE (a < 10 **OR** b < 4)*

NEGATION - Boolean flag (negate), as an additional input parameter to the filter operation. If true, flip the bitmap produced by the evaluate operator keeping rest of operators unchanged

negate(**negate**[a < 10] **AND** **negate**[b < 4])

This approach supports disjunctions and negations with **negligible overhead.**

SELECTION PUSHDOWN IN PARQUET

Adapt previously discussed generic techniques to **Parquet**

```
DocId: 10                 r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20                 r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Figure 2: Two sample nested records and their schema

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Figure 3: Column-striped representation of the sample data in Figure 2, showing repetition levels (r) and definition levels (d)

Definition levels specify how many optional fields in the path for the column are defined.

Repetition levels specify at what repeated field in the path has the value repeated.

The Issue

The challenge arises from the way that Parquet encodes the structure information to represent optional, nested, or repeated fields

Parquet never explicitly stores null values, all repeated values are stored contiguously in the same array.

The number of levels or values in the column may not be the same as the number of records - **fast select operation** cannot be directly applied to Parquet.

PARQUET-SELECT

Authors built a library **Parquet-Select** with the full implementation of the various techniques we discussed so far - BMI, predicate pushdown, fast select etc.

Supports Arbitrary filters - Each filter - UD Lambda function that can implement even complex predicates such as complex string matching, UDFs, cross-table predicates etc.

1. SELECT *FROM products WHERE product_name LIKE '%keyword%' OR product_description LIKE '%keyword%';
2. SELECT product_id, product_name, calculate_discount(product_price) AS discounted_price FROM products;
3. SELECT o.order_id, o.order_date, c.customer_name, c.email FROM orders o JOIN customers c ON o.customer_id = c.customer_id WHERE c.customer_id = 123;

EVALUATION

Processor	2.6GHz AMD EPYC 7413 and Intel Xeon Gold 6140
Memory	256GB DDR4
Storage	NVMe SSD - 3.5GB/s
OS	64-bit Windows Server 2022 Datacenter

Parquet-Select vs open-source C++ version of **Arrow/Parquet2** (v 8.0.0).

- Parquet implements the SIMD-based unpack algorithm for decoding.
- Authors implemented the filter and selection operations on decoded column values using **SIMD** instructions and optimized these operations
- All experiments were run using a single thread

Selection Performance

Compare Parquet-Select and Parquet on a project operation with a given select bitmap - evaluate effects of bit width of the column values and selectivity of the select bitmap.

Parquet file with a single column

- **128 million** 64-bit - Integer values.
- $\mathbf{2^{k}}$ distinct values in the column
- $\boldsymbol{k}$ - bit width of the values encoded with dictionary encoding

This is varied in the experiment

Important Results

1. Parquet-Select slightly better than Parquet when s = 1/1 - selection on bit-packed values read slightly less than selection on decoded values - even with SIMD vectorization - **unpacking** still bottleneck
2. PS - Higher performance gains on decrease bit-width - due to **high data parallelism** in processing bits by fast-select operator

Upto **10X** speedup

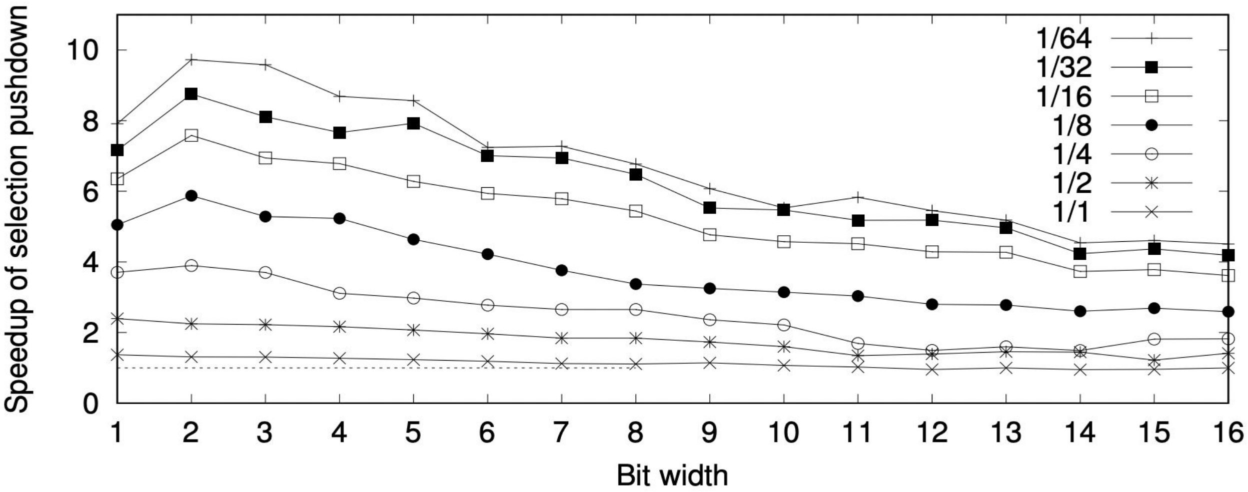


Fig. 12. Parquet vs. Parquet-Select: selection operation

Micro-benchmark Evaluation

Evaluate 2 approaches with scan queries using micro-benchmark

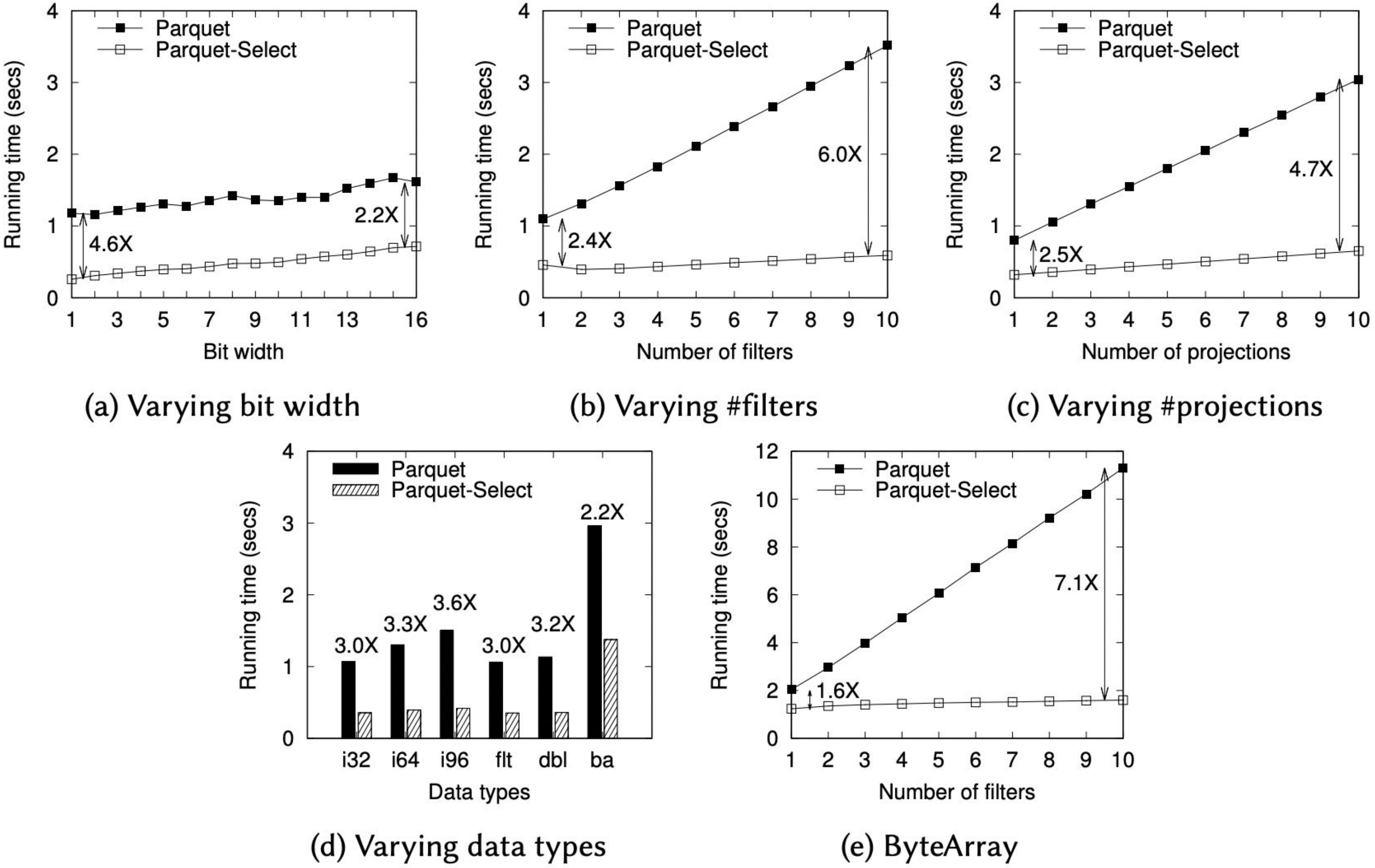
Experiment parameters

- 20 columns (a1 ~ a20)
- 128 million rows
- Selectivity of each filter = 25%
- Overall selectivity - varies as number of filters is varied

SELECT MAX(a10), MAX(a11), ... FROM R
WHERE a1 < C1 AND a2 < C2 AND ...

Evaluation parameters - how performance is affected by

1. Number of filters
2. Number of projections
3. Bit width of column values
4. Data types



(a) Varying bit width (b) Varying #filters (c) Varying #projections

(d) Varying data types (e) ByteArray

Fig. 13. Parquet vs. Parquet-Select: scan micro-benchmark

TPC-H BENCHMARK

Test Parquet-Select with realistic workload - TPC-H

The TPC-H is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance.

It comes with various data set sizes to test different scaling factors -

- TPCH_SF1: Consists of the base row size (several million elements).
- **TPCH_SF10**: Consists of the base row size x 10 - **used in the paper**
- TPCH_SF100: Consists of the base row size x 100 (several hundred million elements).
- TPCH_SF1000: Consists of the base row size x 1000 (several billion elements).

TPC-H Q6

Forecasting Revenue Change Query (Q6) - a **scan query**

This query quantifies the amount of revenue increase that would have resulted from eliminating certain company wide discounts in a given percentage range in a given year. Asking this type of "what if" query can be used to look for ways to increase revenues.

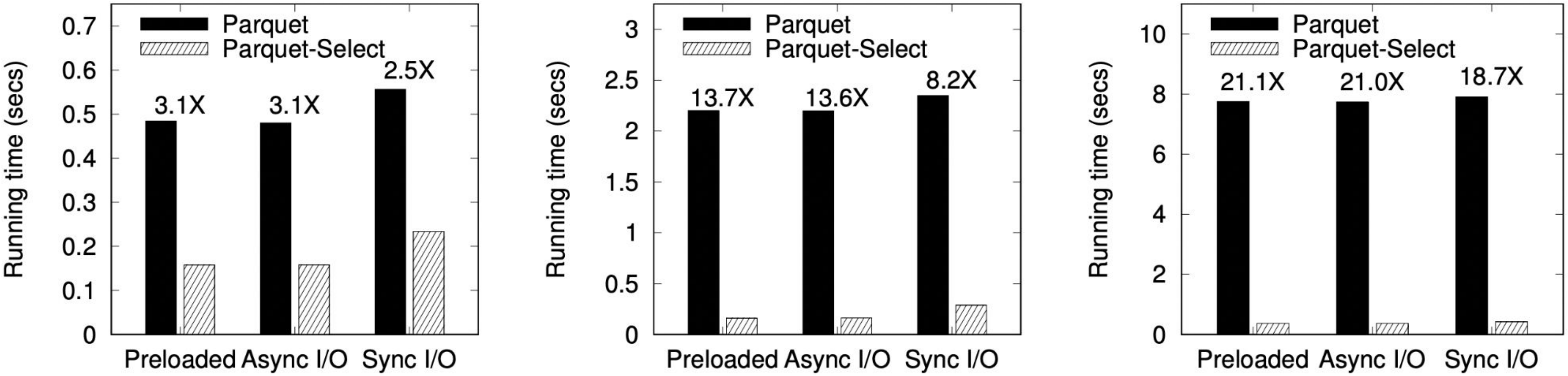
The overall selectivity of the query is **1.9%**

Filters - 3

Columns/Property	l_shipdate	l_discount	l_quantity
Encoding bits	12	4	6
Selectivity	15.2%	27.3%	46.0%

Projection - 2

Columns l_extendedprice and l_discount.



(a) Non-nullable columns

(b) Nullable columns

(c) Repeated columns

Fig. 14. Parquet vs. Parquet-Select: TPC-H benchmark Q6

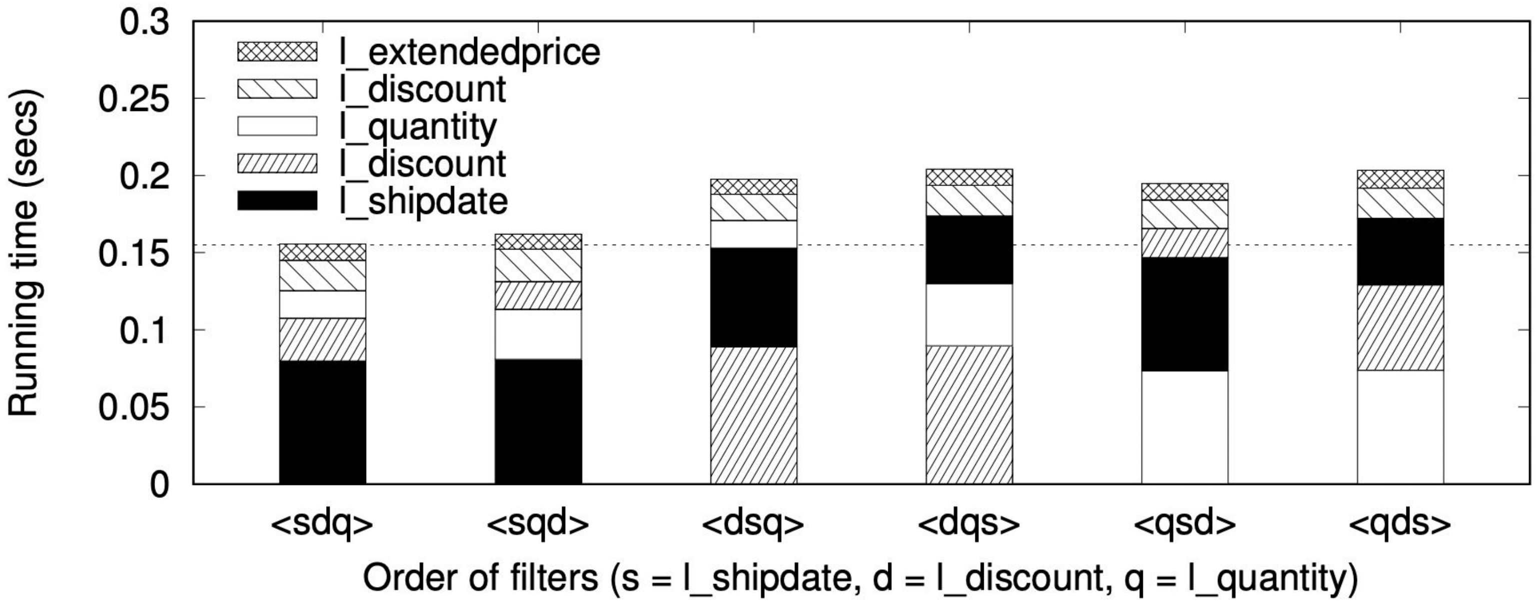


Fig. 15. Impact of filter ordering on TPC-H Q6

Filters in order <sdq> faster than others by **30%**

Why? - Because l_shipdate filter has more selectivity (by **15.2%**) when compared to other filters

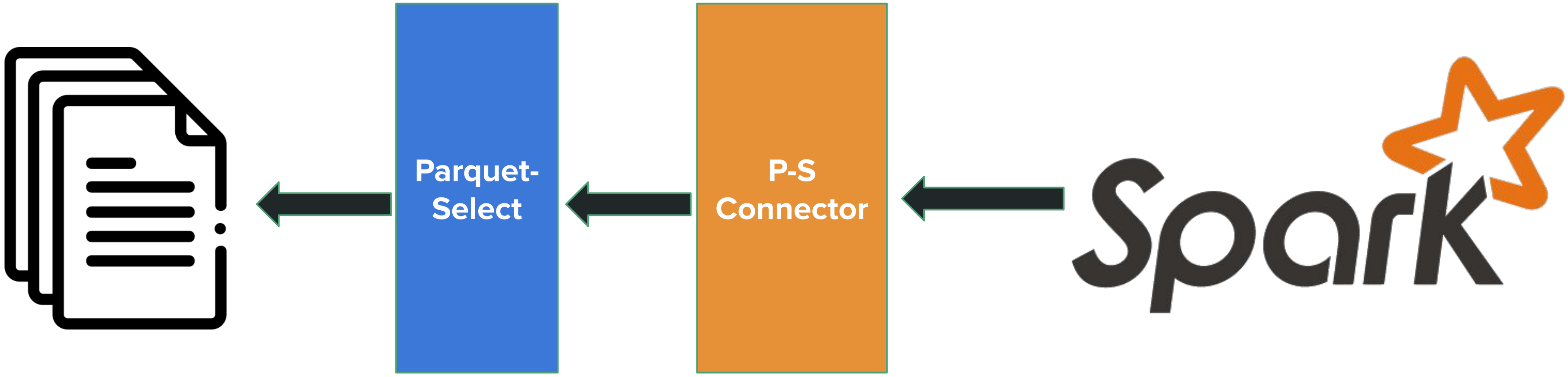
Also size of data in s is the largest - less beneficial to evaluate it later

TPC-H Benchmark Evaluation

Extend evaluation using **Apache Spark -** two approaches

1) **Spark + Parquet** - Standard and deeply integrated way to query Parquet files using Spark
2) **Spark + Parquet-Select** - Enable predicate pushdown in Spark by leveraging Parquet-Select

Using PS in Spark



Parquet Files

Predicate Pushdown

Spark Datasource V2 API

Queries Used for TPC-H Evaluation

Selectivity for each query

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%

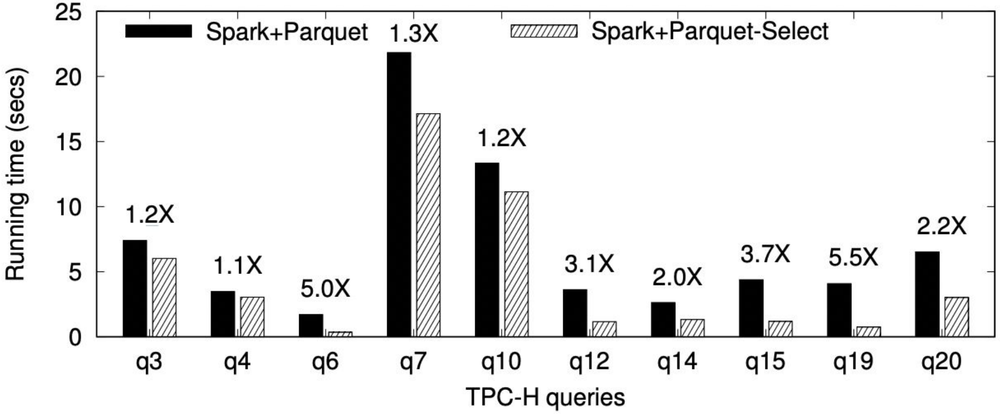


Fig. 16. Spark+Parquet vs. Spark+Parquet-Select: TPC-H

ENCODING AND COMPRESSION

Run-Length Encoding

Apart from bit-pack encoding, other encoding schemes like Run-Length Encoding (RLE) are used for testing their efficiency with selection pushdown for **RLE**-encoded data.

RLE - A simple lossless data-compression where runs of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run

Example

WWWWWWWWWWWW B WWWWWWWWWWWW BBB

12W 1B 12W 3B

Effect of using RLE

Parquet-Select achieves up to 9X speedup over Parquet - as it decodes and selects values in a single pass, while Parquet requires two separate passes to complete these steps

Selection pushdown is generally more efficient for RLE-encoded values than for bit-packed values.

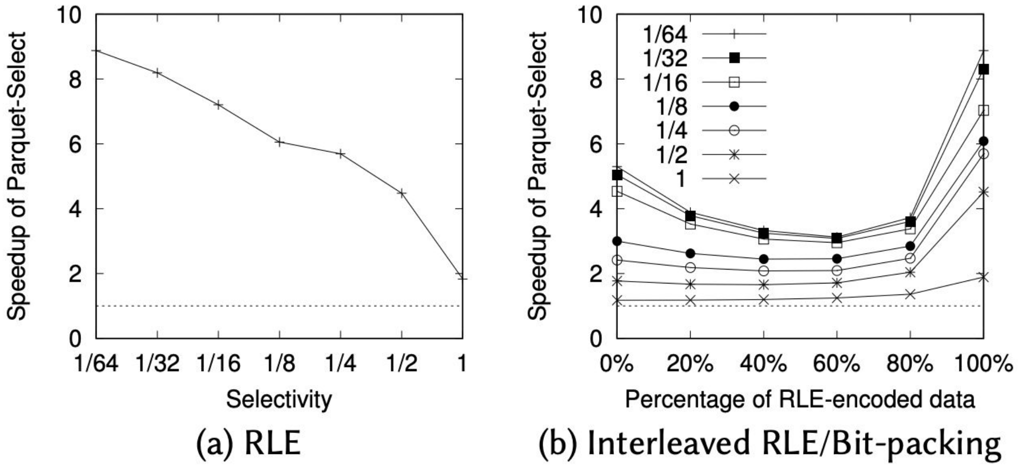


Fig. 17. Impact of RLE encoding

Interleaving bit-packing and RLE

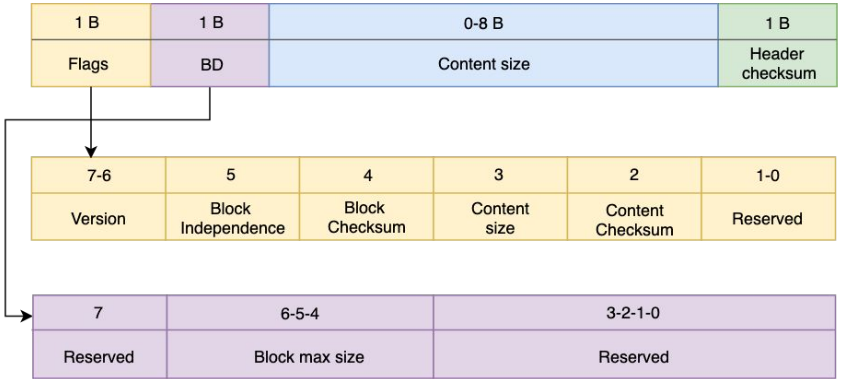
Better to use either the one or the other - don’t interleave RLE and bit-packing

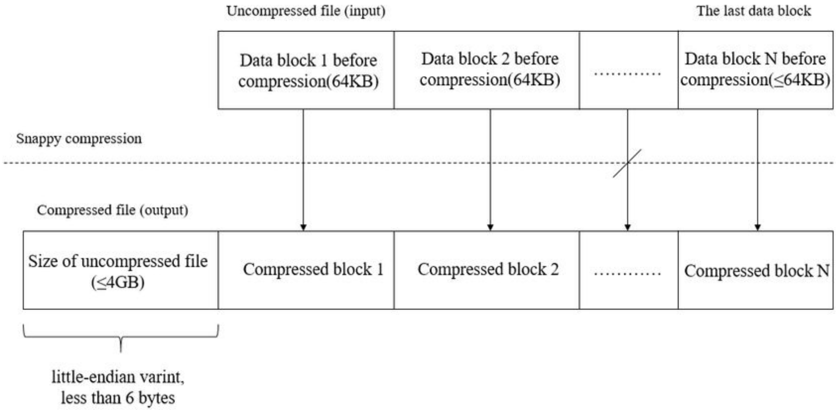
Page-level Compression

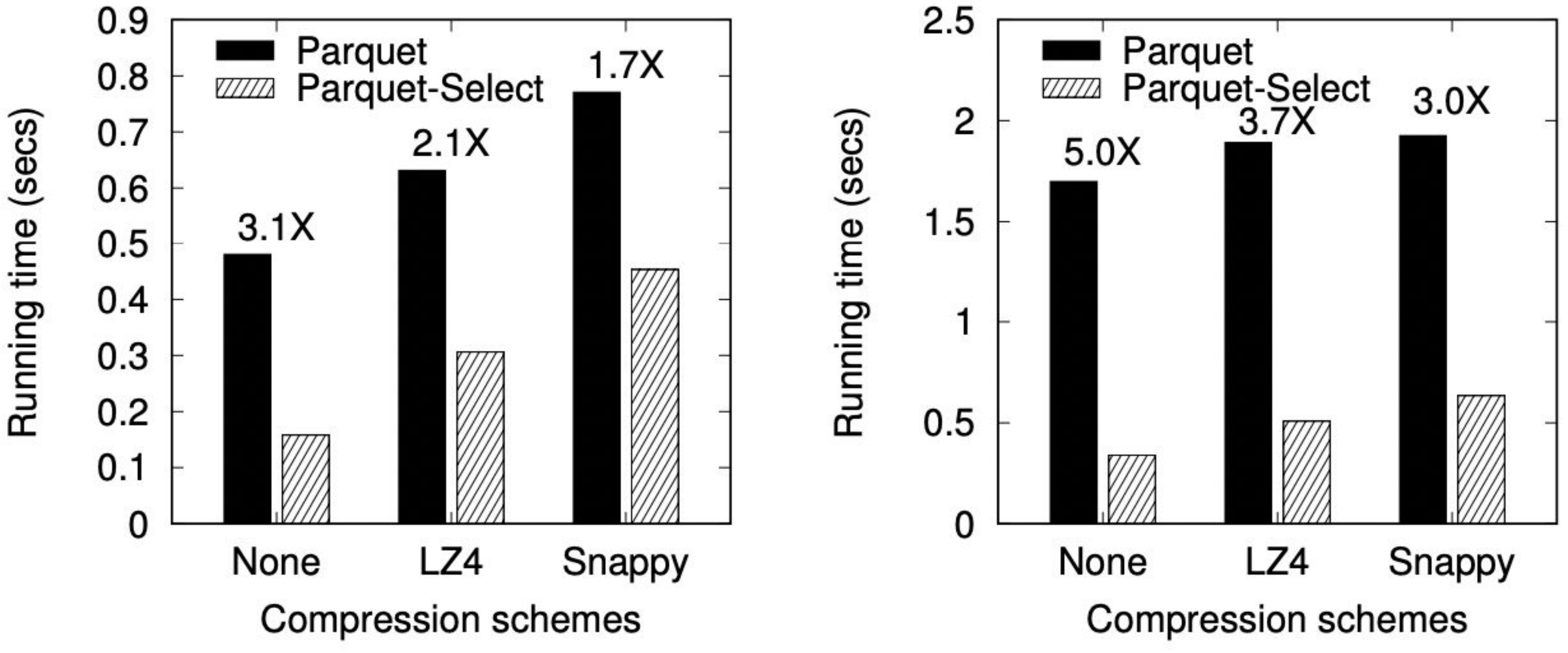
In Parquet, encoded values are organized into data pages, which can be further compressed optionally using a variety of compression schemes.

Here - impact of page-level compression on the performance is done - comparison between Parquet and Parquet-Select using TPC-H Q6

Compression schemes - LZ4 and Snappy - both reduced the size of the Lineitem Parquet file by approximately **30%**







(a) Standalone (b) Spark

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

ANY QUESTIONS?