CSE 350

Advanced Data Structures

Topic 14: Column Stores

Example Queries

SELECT type, address FROM Permits
SELECT city, sum(price) FROM Permits
SELECT type, sum(price) FROM Permits WHERE city = 'Buffalo'

2 attrs/columns

3

[I only need 2-3 cols
I need to load all data]

↖ but I don't know which cols are needed for any specific query

Tuple Linearization

fields

records

a	b	c
d	e	f
g	h	i
j	k	l

[a b c][d e f][g h i] ...

row-store

a	b	c
d	e	f
g	h	i
j	k	l

[a d g j] [b e h k] [c f i l]

column-store

Column Storage (Naive)

Permits(id, type, address, city, price, ...)

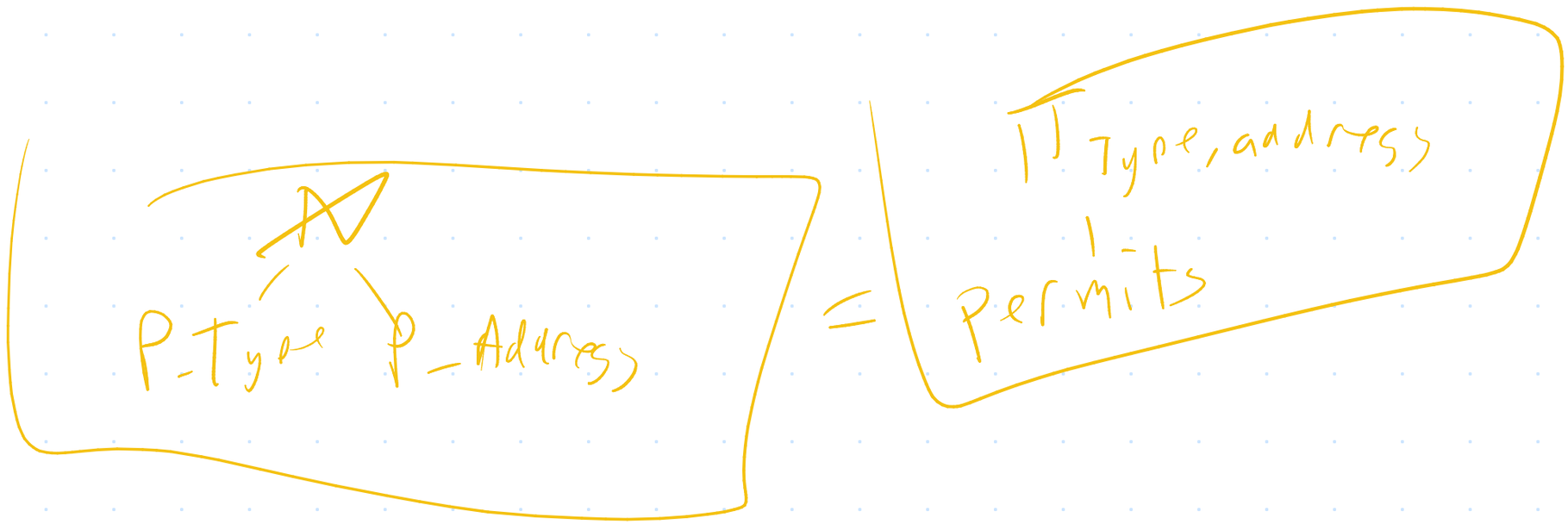
Permits_id(rowid, id)

Permits_type(rowid, type)

Permits_address(rowid, address)

Tuple Reconstruction Join

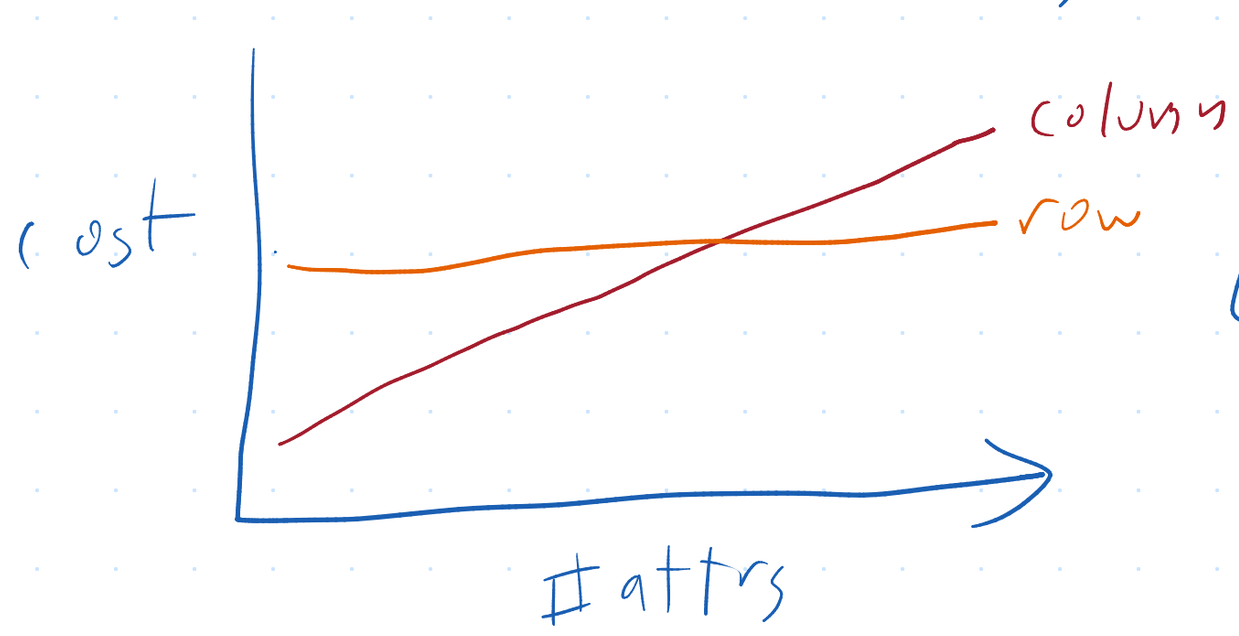
SELECT id, type, address, ...
FROM permit_id
JOIN permit_type ON rowid
JOIN permit_address on rowid
= Permits



Con: Cost grows w/ # of attributes
↳ cost w/ all attrs is higher
↳ CPU cost goes up due to join

More data stored
↳ need to store rowid too

Update cost grows (one IO per attr)
↳ (worse w/ some optimizations)



Variation 1: Sort Order

permits_Type sorted on type
permits-Address sorted on address

can have clustered indexes on multiple attributes

Pro:

Con

permits_city sorted on city
permits_price sorted on price

tuple reconstruction gets much more expensive

↳ solution: store 2 copies of each table
- one sorted on attr
- one sorted on rowid

] more storage (storage is cheap) no more IO needed

---

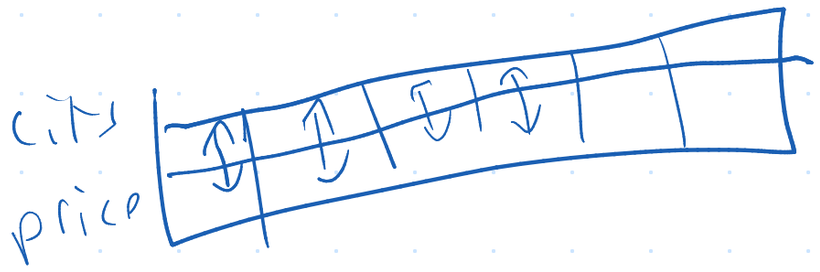
permits_city sorted on city
permits_price sorted on city
] same sort order

↓

correlated

↕

Join becomes a merge (cheaper)



Extreme case: all attrs are correlated & rowid not needed

Implicit key/rowid

visual case mix of both

- ↳ city, address, zip all correlated
  - ↳ rowid is implicitly position in the table
- ↳ all other attrs stored in their own order
  - ↳ rowid stored explicitly in the table

explicit rowid

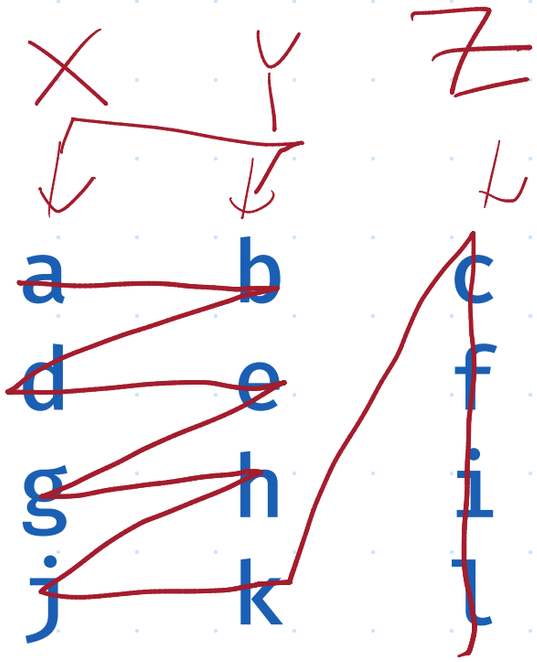
Summary

- Sort order per column
- implicit keys (can't sort in multiple ways)
  - correlated columns
    - faster tuple reconstruction

(can't do both always)

Variation 2: Mix and Match

Column Groupings



[a b d e g h j k c f i l]

XY (rowid, X, Y)

Z (rowid, Z)

SELECT X, Y
FROM ...

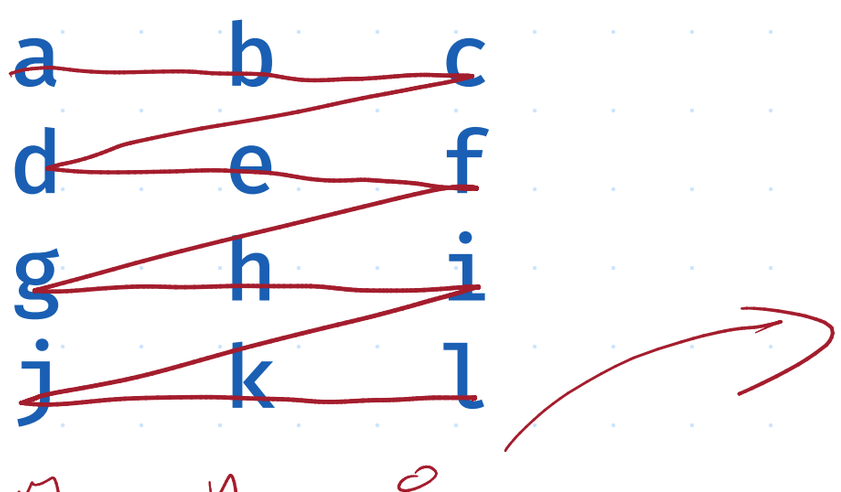
great
No tuple reconstruction

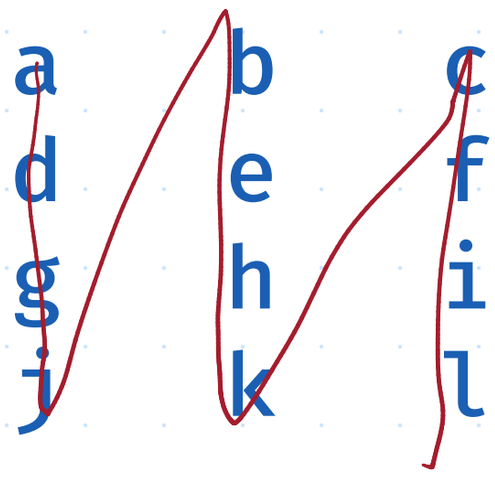
SELECT X
FROM ...

better than row-oriented

but unnecessary (usually)

Fractured Mirrors



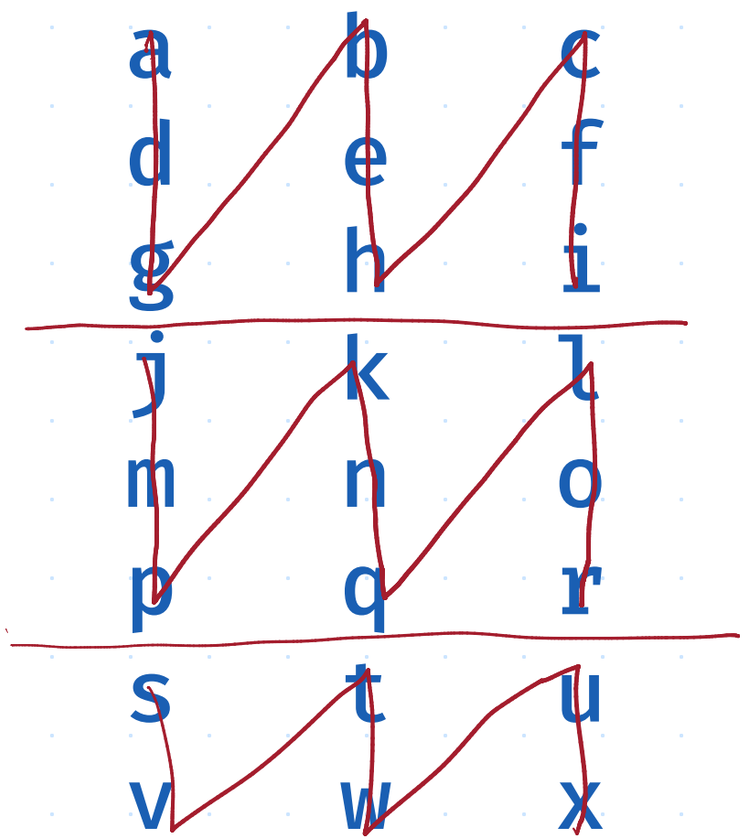


m n o

One copy (ground truth) in row oriented
one copy in column oriented

used in many major DBs

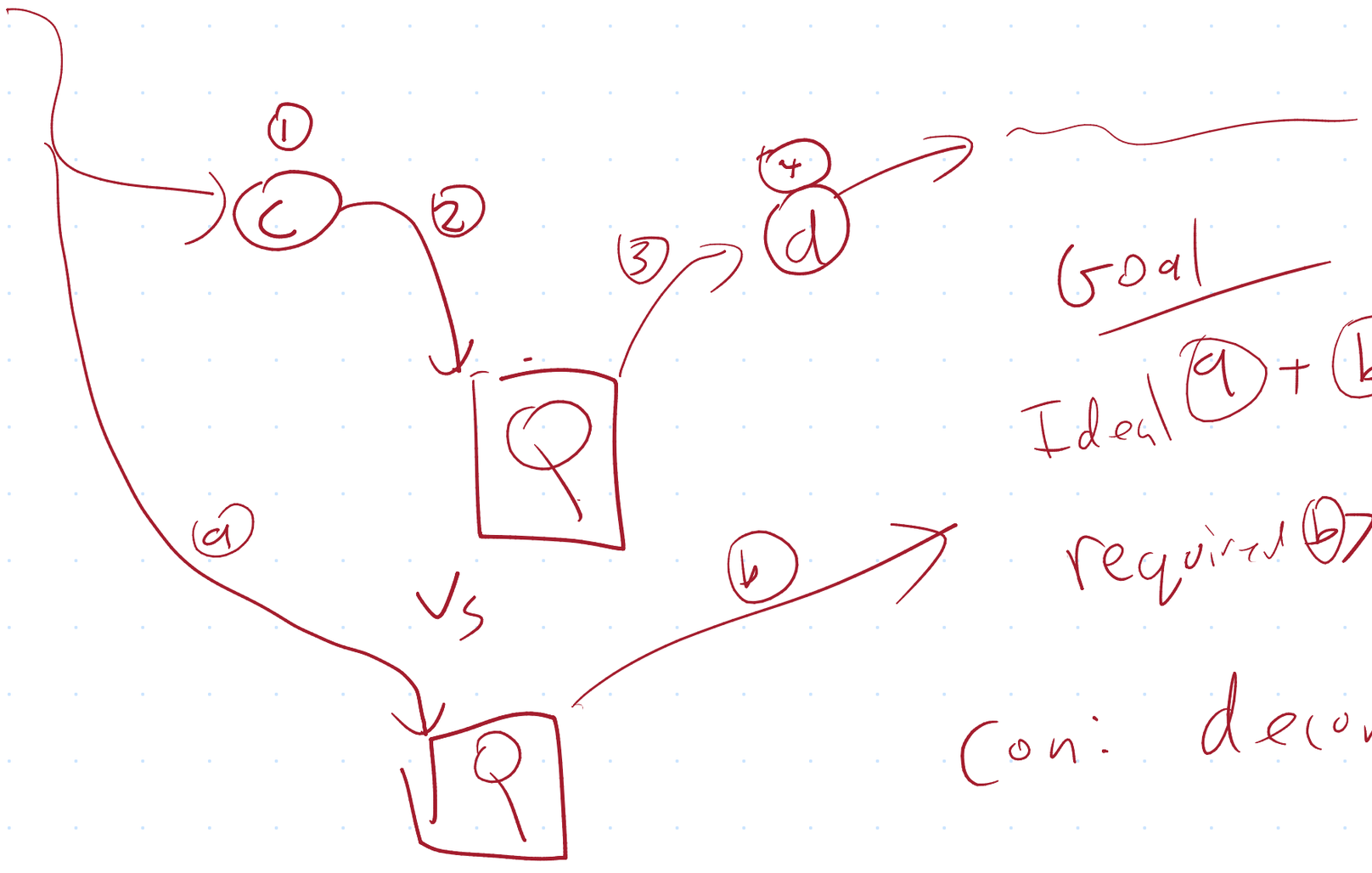
PAX/Decomposed Storage



a d g b e h c f i j m p k n q ...

Compression

```
Permits_Type(rowid, type)
Permits_Address(rowid, address)
```



Goal

Ideal ⓐ + ⓑ > ①+② + ③ + ④

required ⓐ > ③ + ④

Con: decompression is not free

pro: less IO (hopefully)

Compression

X — Data
Y = c(X) — Compression
X = d(Y) — Decompression

$|Y| < |X|$

Examples
Lempel-Ziv
Huffman Coding
DEFLATE (gzip)

Value Compression

Store c(X)

e.g. address is low entropy
↳ good compression ratio
$|c(x)| << |X|$

Dictionary Encoding

rowid	Type
0	USE
1	ELECTRICAL
2	ELECTRICAL
3	PLAN-COPY
4	ELECTRICAL
5	PLUMBING
6	ELECTRICAL
7	ELECTRICAL
8	ELECTRICAL
9	ASBESTOS
10	ELECTRICAL
11	HEATING
12	ELECTRICAL
13	PLAN-COPY
14	HEATING
15	PLUMBING
16	ELECTRICAL
17	REPAIR
18	ELECTRICAL
19	PLAN-COPY

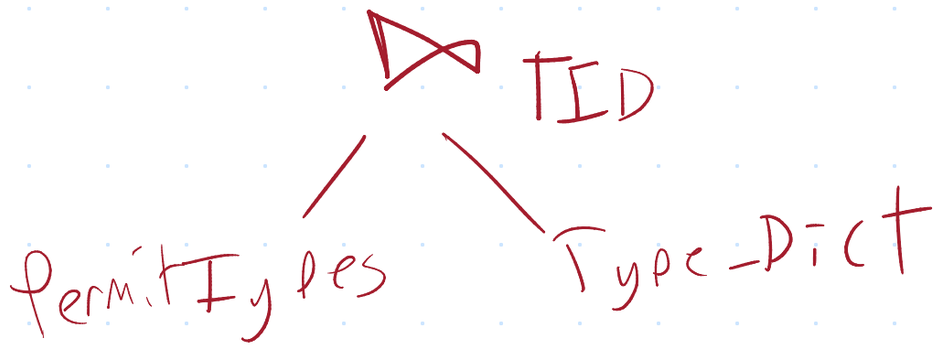
TID

0 1 1 2 1 3 1 1 1

'0' '1' '1' '2' '1' '3' '1' '1' '1'

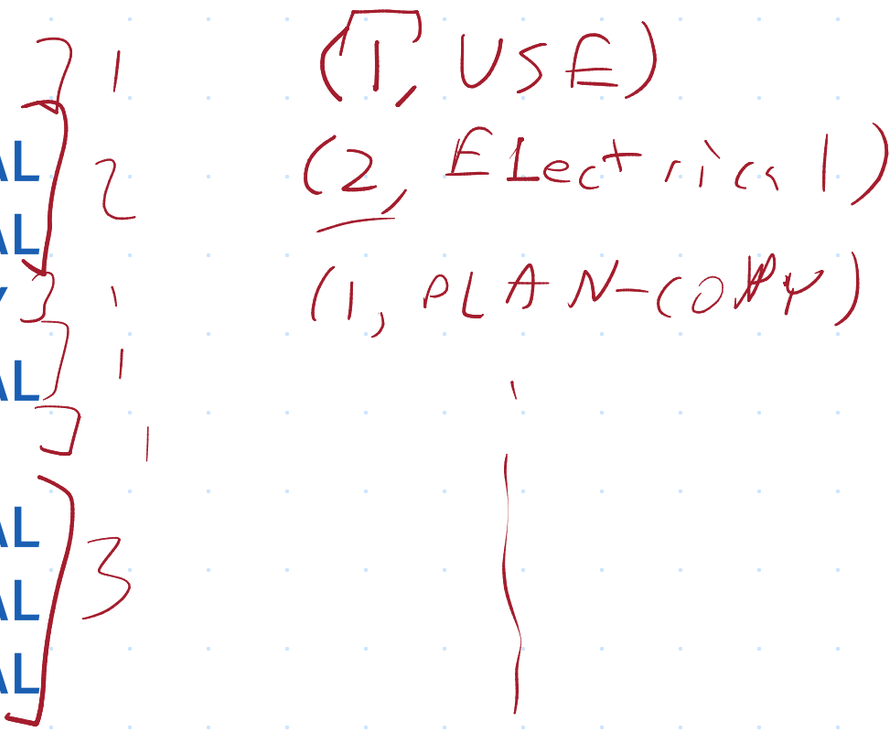
Type_Dict

Type	TID
USE	0
ELECTRICAL	1
PLAN-COPY	2



Run-Length Encoding

rowid	Type
0	USE
1	ELECTRICAL
2	ELECTRICAL
3	PLAN-COPY
4	ELECTRICAL
5	PLUMBING
6	ELECTRICAL
7	ELECTRICAL
8	ELECTRICAL
9	ASBESTOS
10	ELECTRICAL
11	HEATING
12	ELECTRICAL
13	PLAN-COPY
14	HEATING
15	PLUMBING
16	ELECTRICAL
17	REPAIR
18	ELECTRICAL
19	PLAN-COPY



7-Bit Encoding

```
92      0101 1100
10399 0010 1000 1001 1111
```