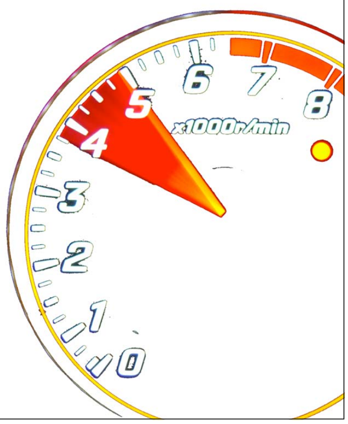
Pipelined Query Execution

Chapter 5



Database Workloads

- Superscalar CPUs can perform multiple instructions in parallel—if enough *indepedent* work is available at a time.
- Query-intensive database workloads like decision support, OLAP, data mining, multimedia retrieval require lots of independent calculations.
- Such workloads thus should provide plenty of opportunity to achieve near-optimal CPI (< 1).

TPC-H Query 1

```
SELECT   l_returnflag, l_linestatus,
         sum(l_quantity) AS sum_qty,
         sum(l_extendedprice) AS sum_base_price,
         sum(l_extendedprice * (1 - l_discount))
           AS sum_disc_price,
         sum(l_extendedprice * (1 - l_discount) *
             (1 + l_tax)) AS sum_charge,
         avg(l_quantity) AS avg_qty,
         avg(l_extendedprice) AS avg_price,
         avg(l_discount) AS avg_disc,
         count(*) AS count_order
FROM     lineitem
WHERE    l_shipdate <= date('1998-09-2')
GROUP BY l_returnflag, l_linestatus
```


High CPI for DBMSs

- Research has shown that DBMSs tend to require high CPI (typically > 2.5) even on modern CPUs, while SPECint programs achieve 0.5 < CPI < 1.5.
- Basic architectural principles in DBMS software—*e.g.*, tuple-at-a-time query execution—are to blame.
  - The commonly implemented Volcano iterator model does *not* exhibit sufficient parallelism.

Volcano Iterator Model

- Each database operator (relational algebra) implements a common interface:

`open()`
Reset internal state and prepare to deliver first result tuple.

`next()`
Deliver next result tuple or indicate `EOF`.

`close()`
Release internal data structures, locks, *etc.*

- Evaluation is driven by the top-most operator which receives open(), next(), next(), ... calls and propagates.

Volcano Iterator Model

```
                           open()
                 next()    next()    π   payment * 2
                           close()   |
                                     |
                           open()
       next()    next()    next()    σ   age < 25        EOF
                           close()   |
                                     |
                           open()
       next()    next()    next()    R
                           close()       employees
```

Volcano Iterator Model: Nested-Loops Join

```
join.open() {
  lhs.open();
  l = lhs.next();
  rhs.open();
}
```

```
join.close() {
  lhs.close();
  rhs.close();
}
```

```
join.next() {
  do {
    if (l == EOF) return EOF;
    r = rhs.next();
    if (r == EOF) {
      l = lhs.next();
      rhs.close();
      rhs.open();
      continue;
    }
  }
  while (¬ Θ(l,r));
  return <l,r>;
}
```

- Note: Variable `l` is static.

Complex Operator Semantics

- Even basic query operators tend to have quite complex semantics.
  - Only at query time `join(lhs,rhs,Θ)` has complete information about relations `lhs`/`rhs` and predicate Θ, for example:
    - number of columns in `lhs`/`rhs`, attribute types, record offsets (*i.e.*, the schema), and an
    - expression interpreter is needed to evaluate Θ.

MySQL `gprof` Trace

% Time	Calls	# Ins.	IPC	Function
11.9	846M	6	0.64	`ut_fold_ulint_pair`
8.5	0.15M	27K	0.71	`ut_fold_binary`
5.8	77M	37	0.85	`memcpy`
3.1	23M	64	0.88	`Item_sum_sum::update_field`
3.0	6M	247	0.83	`row_search_for_mysql`
2.9	17M	79	0.70	`Item_sum_avg::update_field`
2.6	108M	11	0.60	`rec_get_bit_field_1`
2.5	6M	213	0.61	`row_sel_store_mysql_rec`
2.4	48M	25	0.52	`rec_get_nth_field`
2.4	60	19M	0.69	`ha_print_info`
2.4	5.9M	195	1.08	`end_update`
2.1	11M	89	0.98	`field_conv`
2.0	5.9M	16	0.77	`Field_float::val_real`
1.8	5.9M	14	1.07	`Item_field::val`
1.5	42M	17	0.51	`row_sel_field_store_in_mysql`
1.4	36M	18	0.76	`buf_frame_align`
1.3	17M	38	0.80	`Item_func_mul::val`
1.4	25M	25	0.62	`pthread_mutex_lock`
1.2	206M	2	0.75	`hash_get_nth_cell`
1.2	25M	21	0.65	`mutex_test_and_set`
1.0	102M	4	0.62	`rec_get_1byte_offs_flag`
1.0	53M	9	0.58	`rec_1_get_field_start_offs`
0.9	42M	11	0.65	`rec_get_nth_field_extern_bit`
1.0	11M	38	0.80	`Item_func_minus::val`
0.5	5.9M	38	0.80	`Item_func_plus::val`

Tuple-at-a-time Processing

- The `Item_*` operations are invoked by the **π.**`next()` routine (projection), *i.e.*, separately for each tuple.
  - The function call overhead (ca. 20 cycles) must be amortized over only one operation (*e.g.*, addition).
  - The compiler cannot perform loop pipelining. Iteration is "non-local" but *involves all operators* the query tree.

Full Vertical Fragmentation

C_CUSTKEY	C_NAME	⋯	C_PHONE
100	Alice		221-921
101	Bob		303-272
102	Carol		555-901

OID	C_CUSTKEY
0@0	100
1@0	101
2@0	102

OID	C_NAME
0@0	Alice
1@0	Bob
2@0	Carol

OID	C_PHONE
0@0	221-921
1@0	303-272
2@0	555-901

Binary Association Tables (BATs)

BAT[*oid*, *t*]

head	tail
0@0	*a*
1@0	*b*
2@0	*c*
3@0	*d*
4@0	*e*
5@0	*f*

head	tail
0@0	*a*
1@0	*b*
2@0	*c*
3@0	*d*
4@0	*e*
5@0	*f*

- Typically, column `head` contains dense, ascending OIDs (integers).
- BATs degenerate to 1-dim arrays.
- Positional lookups (offset-based).

BAT Algebra: Fixed Schema, Less Freedom

- Equi-join between two BATs:

  join(BAT[t_1,t_2], BAT[t_2,t_3]) : BAT[t_1,t_3]

  - Schema of input and output relations is fixed.
  - No predicate interpreter required.
  - Complex expressions, *e.g.*, `extprice * (1-tax)`:

```
tmp1 := [-](1, tax);
tmp2 := [*](extprice, tmp1);
```


Column-at-a-Time Processing and Pipelining

- The column-at-a-time operators perform many simple operations in a tight loop. Loop unrolling and pipelining is applicable. Implementation of `[-]`:

```
map_sub_double_val_double_col(
  int n,
  double c,
  double* __restrict__ res,
  double* __restrict__ col1)
{
  for (int i=0; i<n; i++)
    res[i] = c - col1[i];
}
```

TPC-H Query 1 Experiments

- Query execution time for TPC-H scale factor SF = 1 (6M rows in table `lineitem`) AthlonMP @1.5 GHz:

MySQL 4.1	MonetDB/ MIL	Hand-coded C
28.1 s	3.7 s	0.22 s

- The “ultra-tight” loops in MonetDB suffer from memory bandwidth limits (ca. 500 MB/s, see upcoming chapters).


```
void tpch_query1(
  int n, int hi_date,
  unsigned char* __restrict__ l_returnflag,
  unsigned char* __restrict__ l_linestatus,
  double* __restrict__        l_quantity,
  double* __restrict__        l_extendedprice,
  double* __restrict__        l_discount,
  double* __restrict__        l_tax,
  int* __restrict__           l_shipdate,
  aggr_t1* __restrict__       hashtab)
{
  for (int i=0; i<n; i++) {
    if (l_shipdate[i] <= hi_date) {
      aggr_t1 *entry = hashtab +
                   (l_returnflag[i] << 8) + l_linestatus[i];
      double discount = l_discount[i];
      double extprice = l_extendedprice[i];
      entry->count++;
      entry->sum_qty += l_quantity[i];
      entry->sum_disc += discount;
      entry->sum_base_price += extprice;
      entry->sum_disc_price += (extprice *= (1 - discount));
      entry->sum_charge += extprice * (1 - l_tax[i]);
    }
  }
}
```

MonetDB/X100

- MonetDB/X100, developed at CWI, Amsterdam. Principal architect is Peter Boncz.

  `http://homepages.cwi.nl/~boncz/x100.html`

- MonetDB/X100 applies full vertical fragmentation internally (column storage).
- Columns are processed in chunks (vectors) using Volcano-style iteration. MonetDB/X100 takes care to ensure that all live vectors fit in the CPU cache.

MonetDB/X100 Algebra

- Operates over *n*-ary tables. Internally: column storage.
- `Table`: materialized table, `Dataflow`: pipelined vectors (typical vector size: 2^{10} values, adapatble to cache size)

`Table(ID):Table`
`Scan(Table):Dataflow`
`Project(Dataflow,List<Exp<*>>):Dataflow`
`Aggr(Dataflow,List<Exp<*>>,List<Aggregates>):Dataflow`
`Select(Dataflow,Exp<bool>):Dataflow`

Selection Vectors

	A
0	10
1	5
2	42
3	35
4	6
5	16

`Select(·,A > 10)`

pos
2
3
5

- No data is copied from the selection source—saves memory traffic if source column is wide.
- Other MonetDB/X100 algebra operators need to be aware of selection vectors.

Highly Specialized Primitives

```
map_plus_double_col_double_col(
  int n,
  double* __restrict__ res,
  double* __restrict__ col1,
  double* __restrict__ col2,
  int *__restrict__    sel)
{
  if (sel) {
    for (int j=0; j<n; j++) {
      int i = sel[j];
      res[i] = col1[i] + col2[i];
    }
  } else {
    for (int i=0; i<n; i++)
      res[i] = col1[i] + col2[i];
  }
}
```

- Uses 2-3 issue cycles/tuple (MySQL: 49 cycles).

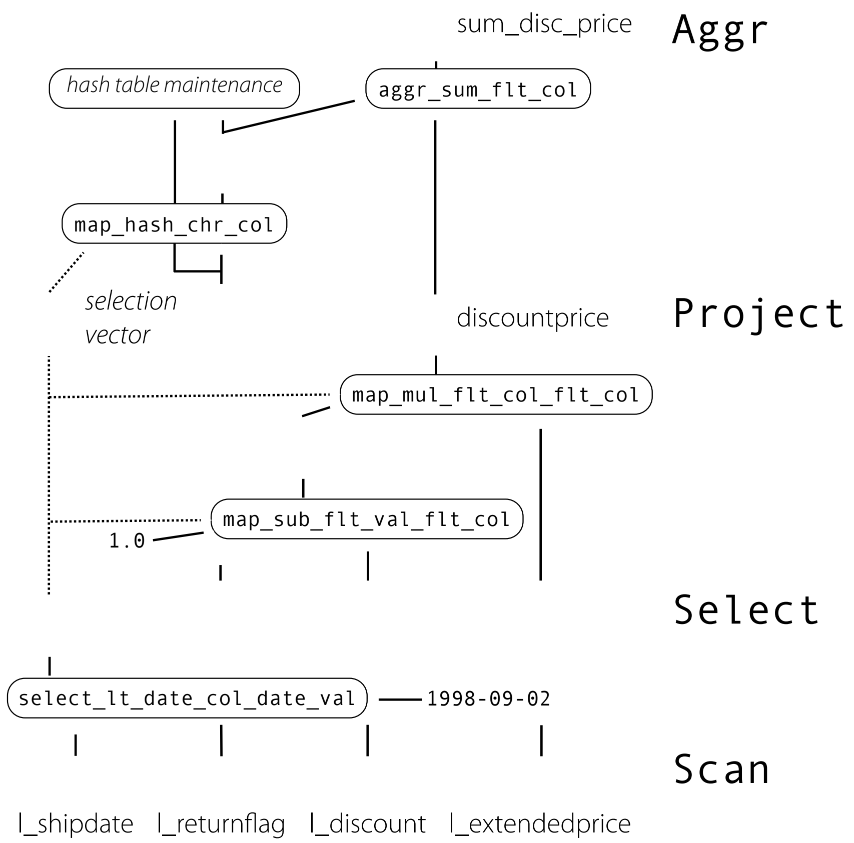
Simplified TPC-H Query 1

SQL

```
SELECT   sum(l_extendedprice * (1 - l_discount))
           AS sum_disc_price
FROM     lineitem
WHERE    l_shipdate < date(“1998-09-02”)
GROUP BY l_returnflag
```

X100 Algebra

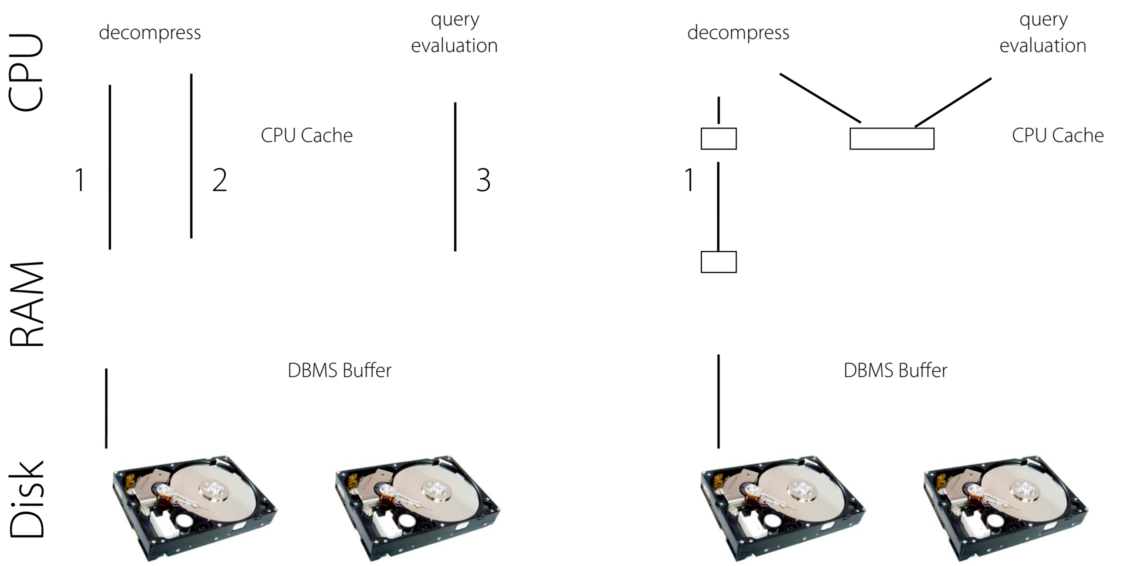
```
Aggr (
  Project (
    Select (
       Scan (Table (lineitem)),
    < (l_shipdate, date(“1998-09-02”))),
   [ discountprice = * (- (1.0, l_discount),
                          l_extendedprice) ])   ,
[ returnflag ],
[ sum_disc_price = sum(discountprice) ])
```

Pipelined Query Evaluation and I/O Hunger

- Vector-based pipelineable query execution leads to extremely high tuple bandwidth figures.
  - If vectors are cache-resident, bandwidths of multiple GB/s are achieved (*e.g.*, 5 GB/s for `map_mult_*` on an Athlon MP @ 1.5 GHz).
  - Modern, high-end RAID systems can only deliver ≈ 0.3 GB/s — Are we hopelessly I/O bound?
- ⇒ Maintain compressed data on disk and RAM.

Disk–RAM vs. RAM–CPU Compression



RAM–CPU Compression

- Avoids to cross the CPU/RAM border 3 times.
  - DBMS buffer manager stores compressed pages and thus can cache more data.
  - Decompress at small granularity (< CPU cache size) and just when the query processor requests it.
- Requires high-bandwidth, lightweight compression schemes.

Decompression Speed

- Bandwidth of generic decompression algorithms will *not* be sufficient:

`bzip2`	`zlib`
10 MB/s	80 MB/s

  - Modern RAID systems deliver 0.3 GB/s.
  - Assume compression ratio of 4:1— decompression will need to sustain a bandwidth of 1.2 GB/s.
  - Invest about 40% of CPU time into decompression: decompression needs to deliver 3 GB/s.

Lightweight Compression: FOR

- Frame of reference (FOR) compressor:
  - Block-wise compression, values $C[i]$. Let min_C denote the minimum C value in the block.
  - In the compressed block, store ($C[i]$ - min_C) values with fixed bit length.
- Requires $\lceil log_2(max_C - min_C + 1) \rceil$ bits per value.

  Works well with clustered data. Also used to compress pointers in B-tree indexes.

Lightweight Compression: DICT

- Dictionary compression (DICT, *enumerated storage*) exploits value distributions which use a small subset of a full domain (= value range admitted by a type).
  - Encode values by a code with minimal bit length:

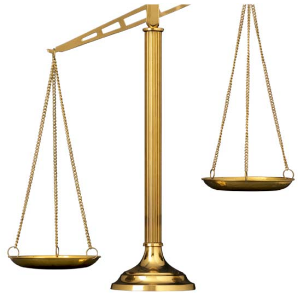
Gender
"female"
"female"
"male"
"female"
"male"

Gender
0
0
1
0
1

+

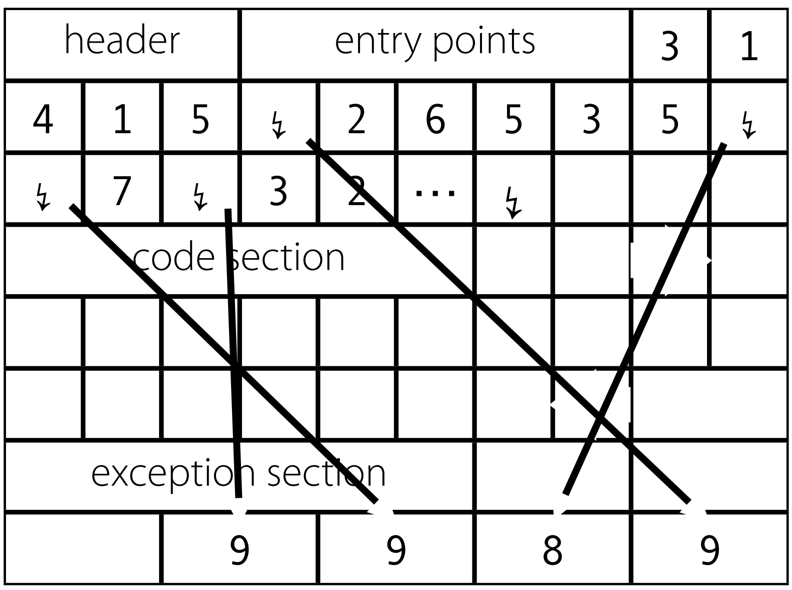
Code	Dict
0	"female"
1	"male"

Skewed Data and Outliers



- FOR and DICT are vulnerable to outliers and skew:
  - FOR suffers from extreme max_C, min_C values.
  - DICT needs $\lceil log_2(|Dictionary|) \rceil$ bits and thus also suffers from skew and outliers.
- Treat outliers as exceptions which do not influence max_C, min_C or the dictionary size, respectively. Requires exception handling.

Block Layout (3-Bit Code): 3.1415926535897932



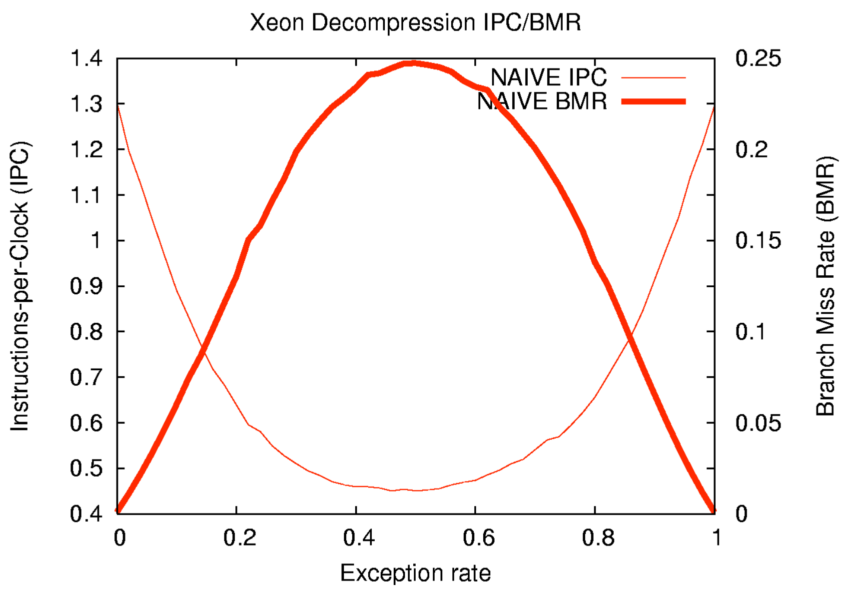
Naïve Decompression

```
int i;       /* points into codes */
int j = 0;   /* points into exceptions */

for (i = 0; i < n; i++) {
  if (code[i] != ϟ)
    output[i] = DECODE(code[i]);
  else
    output[i] = exception[--j]);
}
```

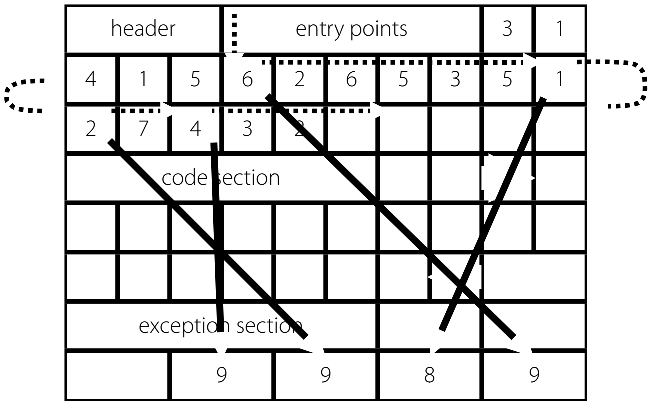
- Assume *b*-bit codes to have been unpacked into array `code[]` (negligible effort).
- Function **`DECODE()`** implements FOR (DICT) decompression.
- Misprediction rate depends on exception ratio.

Naïve Decompression on an Intel Xeon® CPU



Patch Lists

- Maintain a patch list through code word section that links exceptions:
  1. Decode all codes, *including* patch markers.
  2. Patch up the exception positions.



Decompression with Patches

```
void decompress (
  int n;
  int* __restrict__ output,
  int* __restrict__ code,      /* points to code section */
  int* __restrict__ exception, /* points after exceptions */
  int  entry_point             /* first exception */
)
{
  /* phase 1: decode regardless */
  for (int i=0; i < n; i++)
    output[i] = DECODE(code[i]);

  /* phase 2: patch up */
  for (int i=1; entry_point < n; i++) {
    output[entry_point] = exception[-i];
    entry_point += code[entry_point];  /* walk patch list */
  }
}
```

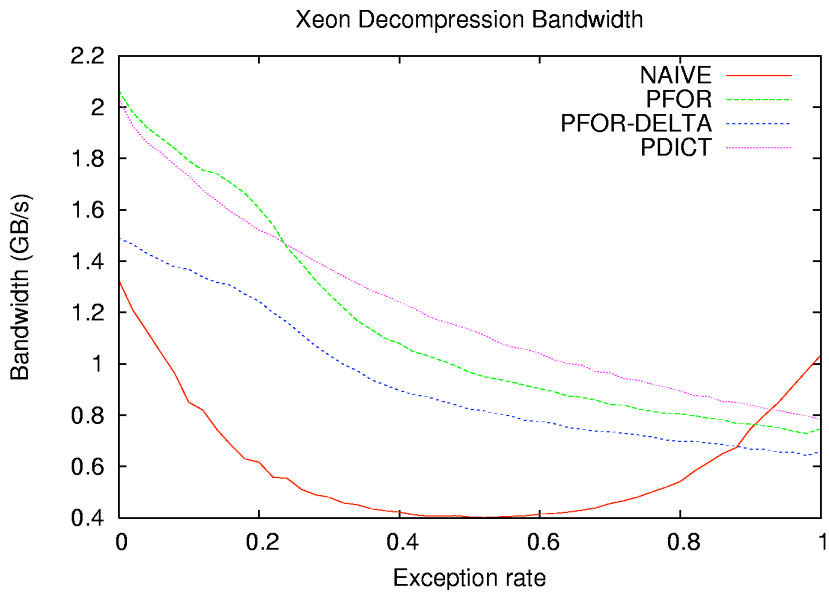
Control Dependence to Data Dependence

- Note the data dependency in the patch loop:

```
/* phase 2: patch up */
for (int i=1; entry_point < n; i++) {
  output[entry_point] = exception[-i];
  entry_point += code[entry_point];  /* walk patch list */
}
```

  - This dependency is inherent to *any* list walking strategy.
  - Data hazards are less costly than control hazards. Patch loop processes small percentage of data only.

Decompression Bandwidth



Compression with Patching

```
void compress (
  int n;
  int* __restrict__ input,
  int* __restrict__ code,      /* points to code section */
  int* __restrict__ exception, /* points after exceptions */
  int* last_patch              /* position of last patch */ )
{
  int miss[N], nexc;

  for (int i = 0,nexc = 0; i < n; i++) {
    int c = ENCODE(input[i]);
    code[i] = c;
    miss[nexc] = i;
    nexc += (c > MAXCODE);      /* MAXCODE = 2^b-1 */
  }
  for (int i = 0; i < nexc; i++) {
    int patch = miss[i];
    exception[-i] = input[patch];
    code[*last_patch] = patch - *last_patch;
    *last_patch = patch;
  }
}
```


Compiling Selection Conditions

- Column-at-a-time selections repeatedly evaluate a given (compound) predicate in a tight inner loop.
- Consider

$$\boldsymbol{\sigma}_{\,p1 \wedge p2 \wedge p3}\,(q)$$

in which we assume predicate p_i to be evaluated on column col_i of the input query q.

Compiling Selection Conditions

```
int j = 0;

for (int i = 0; i < n; i++) {
  if (p1(col1[i]) && p2(col2[i]) && p3(col3[i]))
     res[j++] = i;
  }
```

- In C, && is also known as the *branching and* operator:

```
if (p && q) {
  s;
}
```

compile

```
      …         ; evaluate p (→ R1)
      BEQZ R1,skip
      …         ; evaluate q (→ R2)
      BEQZ R2,skip
      s;              ; code for s
skip: …
```


Compiling Selection Conditions

```
int j = 0;

for (int i = 0; i < n; i++) {
  if (p1(col1[i]) & p2(col2[i]) & p3(col3[i]))
     res[j++] = i;
  }
```

- Operator & performs *bitwise and* (no shortcut eval):

```
if (p & q) {
  s;
}
```

compile

```
      …         ; evaluate p (→ R1)
      …         ; evaluate q (→ R2)
      AND  R3,R1,R2
      BEQZ R3,skip
      s;              ; code for s
skip: …
```

Compiling Selection Conditions

```
int j = 0;

for (int i = 0; i < n; i++) {
     res[j] = i;
     j += (p1(col1[i]) & p2(col2[i]) & p3(col3[i]))
}
```

- In C, Booleans are represented as `0` (*false*) or `1` (*true*):

```
j += p & q;
```

compile

```
…          ; evaluate p (→ R1)
…          ; evaluate q (→ R2)
AND  R3,R1,R2
ADD  R4,R4,R3         ; j ≡ R4
```


Need Cost Model to Select Between Variants

- `p && q`:
  When `p` is highly selective this might amortize the double branch misprediction risk.

- `p & q`:
  Number of branches halved but `q` is evaluated regardless of `p`'s outcome.

- `j += …`:
  Performs memory write in *each* iteration.

Mixed-Mode Selection

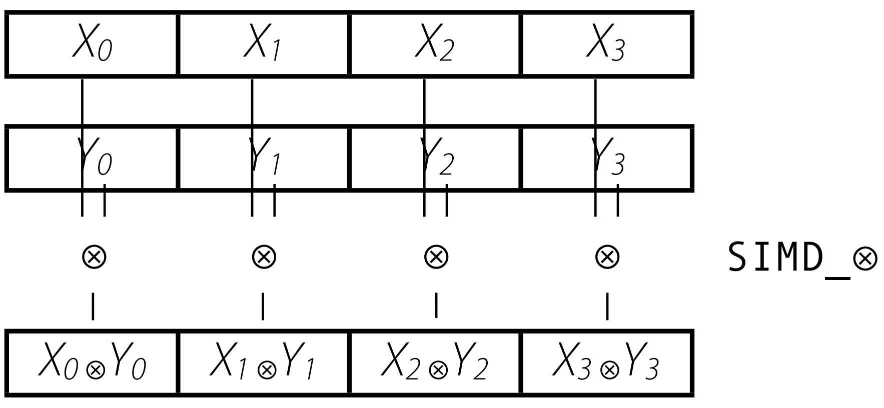
```
int j = 0;

for (int i = 0; i < n; i++) {
     if (p1(col1[i]) & p2(col2[i]) && p3(col3[i])) {
     res[j] = i;
     j += p4(col4[i]))
}
```

- Problem:
  Programming language compiler would need information about database-level meta-data (*e.g.*, selectivities) to make informed choice.
- Enable runtime choice: compile specialized variants, dynamic compilation and linking, self-modifying code.

Exploiting SIMD Operations

- SIMD (single instruction, multiple data) instructions have primarily been added to modern CPUs to accelerate multi-media operations:



SIMD Intrinsics

- Ideally, a programming language compiler would automatically detect opportunities to exploit SIMD instructions (*e.g.*, after loop unrolling).
  - Today's compilers still miss too many (non-)obvious opportunities ⇒ use explicit SIMD intrinsics:

    `SIMD_<`, `SIMD_AND`, `SIMD_+`, ...
- Here: use Pentium4 SSE SIMD intrinsics, 128-bit SIMD registers. Pack `S` = 4 32-bit FP values into SIMD reg.

SIMD: Scan-Like Operations

- High-level structure (assume `N` mod `S` = 0):

```
for (i = 1; i <= N; i++) {
  if (Θ(x[i]))
    process1(y[i]);
  else
    process2(y[i]);
}
```

- High-level structure after introduction of SIMD intrinsics (`SIMD_Process()` needs to be adapted):

```
for (i = 1; i <= N; i += S) {
  mask[1..S] = SIMD_ Θ(x[i..i+S-1]);
  SIMD_Process(mask[1..S], y[i..i+S-1]);
}
```

Return First Match

```
for (i = 1; i <= N; i++) {
  if (Θ(x[i]))
    process1(y[i]);
  else
    process2(y[i]);
}
```

```
for (i = 1; i <= N; i++) {
  if (Θ(x[i]))
    { result = y[i]; return; }
  else
    ;
}
```

```
SIMD_Process(mask[1..S], y[1..S])
{
  int V = SIMD_bit_vector(mask);    /* V ∈ [0,2^S-1] */
  if (V != 0) {
    for (int j = 1; j <= S; j++)
      if ((V >> (S - j) & 1) {
        result = y[j];
        return;
      }
  }
}
```


Return All Matches

```
for (i = 1; i <= N; i++) {
  if (Θ(x[i]))
    process1(y[i]);
  else
    process2(y[i]);
}
```

```
for (i = 1; i <= N; i++) {
  if (Θ(x[i]))
    { result[pos++] = y[i]; }
  else
    ;
}
```

```
SIMD_Process(mask[1..S], y[1..S])
{
  int V = SIMD_bit_vector(mask);    /* V ∈ [0,2^S-1] */
  if (V != 0) {
    for (int j = 1; j <= S; j++) {
      int match = (V >> (S - j)) & 1;
      result[pos] = y[j];
      pos += match;
    }
  }
}
```

Aggregation (`SUM`)

```
SELECT   SUM(R.y)
FROM     R
WHERE    Θ (R.x)
```

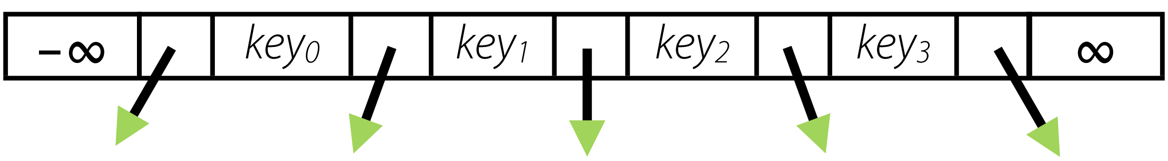
```
for (i = 1; i <= N; i++) {
  if (Θ(x[i]))
    { result += y[i]; }
  else
    ;
}
```

```
SIMD_Process(mask[1..S], y[1..S])
{
  sum[1..S] = SIMD_+(sum[1..S],
                     SIMD_AND(mask[1..S], y[1..S]));
}
```

- Finally, sum up the `S` 32-bit words in `sum[1..S]`.

Search in Internal B+-Tree Nodes

$key_2 \leq K < key_3$



- Common implementation in DBMS: Perform *binary search* (search key *K*) among B+-tree node keys. Determine *branch number* (here: 3).
- B+-tree implementations strive for high fan-out.

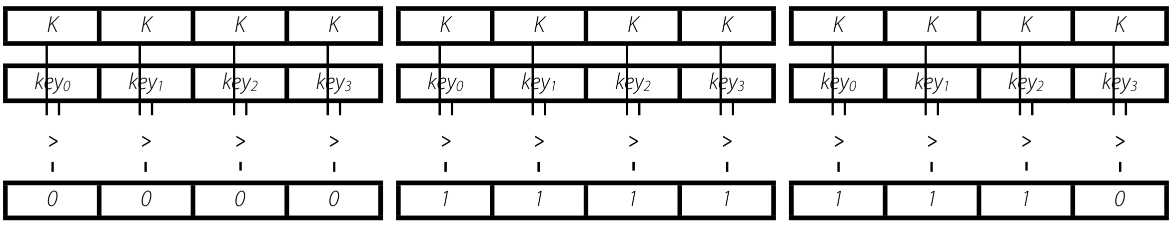
Binary Search and Branch Prediction

```
int bin_search(
  double* keys,
  double  K,
  int     l, r)
{
  int mid = ((unsigned)(l + r)) >> 1;

  if (keys[mid] <= K && K < keys[mid+1])
    return mid;
  if (K >= keys[mid+1])
    return bin_search(keys,K,mid+1,r);
  /* K < keys[mid] */
  return bin_search(keys,K,l,mid);
}
```

- Branch guides the search and will be unpredictable.

SIMD in Binary Search

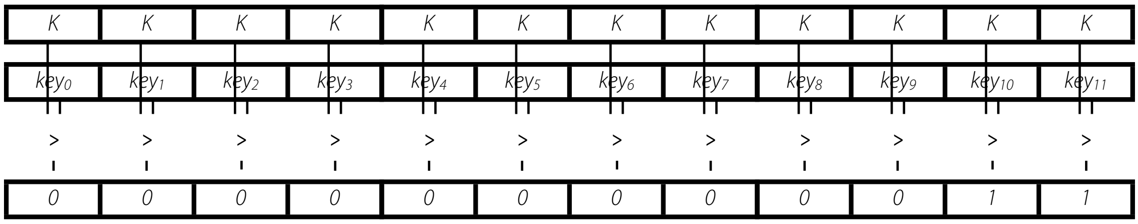


search among keys smaller than *key*$_0$ (left)

search among keys larger than *key*$_3$ (right)

key found, SIMD mask indicates branch number

SIMD in Sequential Search



- Search *sequentially*, left to right.
  Branch number ≡ # of 0 bits in the SIMD masks.
- Avoids (almost all) branches during the search but touches about 50% of all key values in B+-tree node.