hash joins

prof. Stratos Idreos

HTTP://DASLAB.SEAS.HARVARD.EDU/CLASSES/CS165/
give me all students enrolled in cs165

```sql
select student.name from students, enrolled, courses where courses.name="cs165" and enrolled.courseld=course.id and student.id=enrolled.studentld
```
new resL[]; new resR[]; k=0
for (i=0; i<L.size; i=i++)
    for (j=0; j<R.size; j++)
        if L[i]==R[j]
            resL[k]=i
            resR[k++]=j
outer

inner

oracle

search in O(1)
hash join

join input 1 → hash → hash table
hash join

join input 1 -> hash table

join input 2
hash join

on which of the two inputs and when do we build the hash table
val (key), pos

| N keys |
val (key), pos

N keys

hash
val (key), pos

N keys

hash table

hash

k buckets
val (key), pos

hash table

N keys

h = f(val)
bucket = h mod k
val (key), pos

N keys

h = f(val)
bucket = h mod k

[(val1, pos1), (val7, pos7), ...]
val (key), pos

N keys

h = f(val)
bucket = h mod k

[(val1, pos1), (val7, pos7), ...]

hash

hash table

k buckets

bucket size
val (key), pos

N keys

h = f(val)
bucket = h mod k

[(val1, pos1), (val7, pos7), ...]

bucket size
bucket layout
val (key), pos

h = f(val)
bucket = h mod k

[(val1, pos1), (val7, pos7), ...]

bucket size
bucket layout
cost insert/get
val (key), pos

hash table

h = f(val)
bucket = h mod k

[(val1, pos1), (val7, pos7), ...]

should we include pos or actual payload?

bucket size

bucket layout

cost insert/get
how are hash tables different than trees
(leave construction aside for now)
assumption for today:
we know how many buckets we need
(more in next class about this)
goal: join $R(\text{val,pos})$ and $S(\text{val,pos}) = \text{Res}(\text{posR,posS})$
goal: join R(val,pos) and S(val,pos) = Res(posR,posS)

Two columns or “rows”-> same thing, Pure columnar storage gives more flexibility typically MonetDB moved to pure columns instead of rows of column pairs after ~ 10 years (column pair: head, tail) Project: pure columns
goal: join $R(val, pos)$ and $S(val, pos) = Res(posR, posS)$
goal: join $R(\text{val, pos})$ and $S(\text{val, pos}) = \text{Res(pos}_R, \text{pos}_S)$

need one page to stream $R$

Level 1
5 pages

Level 2
$R + S << L2$
goal: join $R(val,pos)$ and $S(val,pos) = Res(posR,posS)$

need one page to stream $R$

need one page to write $Res$

Level 1
5 pages

Level 2
$R+S<<L2$

Res
goal: join $R(\text{val}, \text{pos})$ and $S(\text{val}, \text{pos}) = \text{Res}(\text{posR}, \text{posS})$

need one page to stream $R$

need one page to write $\text{Res}$

hash table, $S <= L1-2$

Level 1
5 pages

Level 2
$R + S << L2$

$R$

$S$

$\text{Res}$
goal: join $R(\text{val,pos})$ and $S(\text{val,pos}) = Res(\text{posR,posS})$

- need one page to stream $R$
- hash table, $S \leq \text{L1-2}$
- need one page to write $Res$

Level 1
5 pages

Level 2
$R + S \ll \text{L2}$
goal: join $R(val, pos)$ and $S(val, pos) = Res(pos_R, pos_S)$
goal: join $R(val,pos)$ and $S(val,pos) = Res(posR,posS)$

need one page to stream $S$

create hash table on $S$

Level 1
5 pages

Level 2
$R+S<<L2$
goal: join $R(\text{val, pos})$ and $S(\text{val, pos}) = Res(\text{pos}_R, \text{pos}_S)$

need one page to stream $S$

create hash table on $S$

I/O Cost in pages = $R + S + Res$
goal: join $R(val,pos)$ and $S(val,pos) = Res(posR,posS)$

need one page to stream $S$

create hash table on $S$

I/O Cost in pages = $R + S + Res$

Same as nested loops cost when small side fits in L1!
what if none of the join inputs fits in L1: R>L1-2 & S>L1-2?

new resL; new resS; k=0; r=0;
for (i=0; i<S.size; i++, k++)
    addToHash(ht, S[i].val, S[i].pos)
    if (k==L1-2 || i==S.size-1)
        for (j=0; j<R.size; j++)
            res = probe(ht, R[i].val);
            if (res!=null)
                resR[r]=R[j].[pos]
                resS[r++]=res.pos
empty(ht)
k=0

hash first L1-2 values from S
scan all R and probe
repeat until we hash everything in S

I/O Cost in pages=S+RxS/(L1-2)+Res
what if none of the join inputs fits in L1: R>L1-2 & S>L1-2?

new resL; new resS; k=0; r=0;
for (i=0, i<S.size; i++, k++)
    addToHash(ht, S[i].val, S[i].pos)
    if (k==L1-2 || i==S.size-1)
        for (j=0; j<R.size; j++)
            res=probe(ht, R[i].val);
        if (res!=null)
            resR[r]=R[j].pos
            resS[r++]=res.pos
    empty(ht)
k=0

hash first L1-2 values from S
scan all R and probe

This is a nested loop-like hash join

RxS factor will be an issue as data grows...

I/O Cost in pages=S+RxS/(L1-2)+Res
Computer science solution space:

*abstractions*

and at a lower level:

*caching, partitioning, indirection,..., not many more*
grace hash join

join input 1

join input 2

partitioning (hash or range)
grace hash join

partitioning (hash or range)

Step 1:
one pass to partition each input
join input 1  join input 2

grace hash join

partitioning (hash or range)

D1

All values in domain \( D1=[\text{min1-max1}] \)

Step1:
one pass to partition each input
grace hash join

join input 1

join input 2

partitioning (hash or range)

D1

D2

All values in
domain D1=[min1-max1]

All values in
domain D2=[min2-max2]

Step 1:
one pass to
partition each input
The grace hash join algorithm involves the following steps:

1. **Join Input Preparation**
   - **Join Input 1**: All values in domain \( D_1 = [\text{min1} - \text{max1}] \)
   - **Join Input 2**: All values in domain \( D_2 = [\text{min2} - \text{max2}] \)
   - **Join Input 3**: All values in domain \( D_3 = [\text{min3} - \text{max3}] \)

2. **Partitioning**
   - The inputs are partitioned using either hash or range partitioning.

3. **Step 1**
   - One pass to partition each input.
grace hash join

join input 1  

join input 2  

grouping (hash or range)

Step 1:
one pass to partition each input

D1

All values in domain D1=[min1-max1]

D2

All values in domain D2=[min2-max2]

D3

All values in domain D3=[min3-max3]

D4

All values in domain D4=[min4-max4]
grace hash join

Step 1:
- one pass to partition each input
Grace hash join

Step 1:
One pass to partition each input

Join input 1

Join input 2

Partitioning (hash or range)
grace hash join

Step 1:
one pass to partition each input

join input 1

join input 2

partitioning (hash or range)
Join input 1 \[\text{join input 2}\]

Partitioning (hash or range)

**Step 1:**
- One pass to partition each input

**Step 2:**
- Then one more pass to join each pair of partitions independently in L1

Grace hash join
grace hash join

join input 1

join input 2

d1
d2
d3
d4

d1
d2
d3
d4

partitioning (hash or range)

**Step1:**
- one pass to partition each input

**Step2:**
- then one more pass to join each pair of partitions independently in L1
grace hash join

partitioning (hash or range)

join input 1

join input 2

Step 1:
one pass to partition each input

Step 2:
then one more pass to join each pair of partitions independently in L1
grace hash join

Step 1:
one pass to partition each input

Step 2:
then one more pass to join each pair of partitions independently in L1

join input 1
join input 2

partitioning (hash or range)

nothing to do here
Grace Hash Join

Step 1: one pass to partition each input

Step 2: then one more pass to join each pair of partitions independently in L1
I/O Cost?

Step 1:
- One pass to partition each input

Step 2:
- Then one more pass to join each pair of partitions independently in L1

D1
D2
D3
D4

nothing to do here
**I/O Cost?**

I/O cost in pages = 3(R+S) + res

So now this scales with data size

---

**Step 1:**
One pass to partition each input

**Step 2:**
Then one more pass to join each pair of partitions independently in L1

---

D1

D2

D3

D4

nothing to do here

---

R+S for first pass
R+S to write partitions
R+S for second pass
result
1. read input into stream buffer, hash and write to respective partition buffer
2. when input buffer is consumed, bring the next one
3. when a partition buffer is full, write to L2

we can partition into L1-1 pieces in one pass
as long as at least one of the pieces $\leq L1-2$
join input 1

join input 2

both left and right side >L1-2
apply recursively
if a partition
does not fit in memory
for every L.key = R.key pair
return [L.pos,R.pos]
data/results
one column-at-a-time

1. quick analysis: when can we do grace join in 2 passes (per input) max?
2. redesign grace join to utilize multi-cores
3. keep all cores at 100% all the time (minimize L3 cache misses and no TLB misses)
Browse: **Join Processing in Databases with Large Main Memories**
L. Shapiro
ACM Transactions on Database Systems. 11(3), 1986

Read: **Cache Conscious Algorithms for Relational Query Processing**
A. Shatdal, C. Kant and J. Naughton
Very Large Databases Conference, 1984

Read: **Database Architecture Optimized for the new Bottleneck: Memory Access**
P. Boncz, S. Manegold and M. Kersten
Very Large Databases Conference, 1999

Browse: **Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs**
Changkyu Kim, et al.
International Conference on Very Large Databases, 2009
hash joins

DATA SYSTEMS

prof. Stratos Idreos