# Parallelization on Multi-Core CPUs

# Amdahl's Law

- suppose we parallelize an algorithm using $n$ cores and $p$ is the proportion of the task that can be parallelized ($1 - p$ cannot be parallelized)
- the speedup of the algorithm is

$$\frac{1}{(1-p)+\frac{p}{n}}$$

- assuming infinite parallelism, the speedup is

$$\frac{1}{(1-p)}$$

- for example, if 90% of the work is parallelized, the maximum speedup is only 10
- one should make sure that every phase of one's algorithm that depends on the input data size is parallelized

# Parallelization Constructs and Libraries

- low-level: C++ threads, pthreads (threads, mutexes, barriers, condition variables)
- parallel patterns: parallel reduce, parallel for, fork/join parallelism
- parallel frameworks: TBB, OpenMP, Cilk Plus

# Intel Thread Building Blocks

- Open Source library for parallelism and concurrency
- fairly nice for prototyping
- manages a pool of worker threads
- implements work stealing
- provides high-level abstractions
- enables nested parallelism
- large systems (e.g., database systems) will have their own framework

# Thread-Local Storage

- in C++ variables can be annotated as `thread_local` (each thread has its own copy)
- however, sometimes it would be convenient to access the thread-local state of other threads
- `tbb::enumerable_thread_specific` allows this

# Parallel Reduce

```
tbb::parallel_reduce(
   tbb::blocked_range<uint64_t>(0, n), // range
   0ull, // identity
   [&](const tbb::blocked_range<uint64_t>& r, uint64_t init) {
      // accumulate
      for (uint64_t i=r.begin(); i!=r.end(); i++)
         init += array[i];
      return init;
   },
   [] (uint64_t x, uint64_t y) { return x+y; }); // combine
```

```
tbb::blocked_range(Value begin, Value end, size_type grainsize=1);
```

# Parallel For

```
tbb::parallel_for(tbb::blocked_range<uint64_t>(0, n),
   [&](const tbb::blocked_range<uint64_t>& r) {
      for (uint64_t i=r.begin(); i!=r.end(); i++)
         array[i] *= 2;
    });
```

# Partitioners

- `parallel_for` and `parallel_reduce` split the given range to enable parallel execution
- there are multiple builtin partitioners:
  - `static_partitioner` splits work equally among threads up-front (no dynamic work stealing)
  - `simple_partitioner` splits the range as much as possible (e.g., until `grainsize` is reached)
  - `auto_partitioner` heuristic similar to `simple_partitioner`, but tries to avoid creating too many ranges (default)

# Fork/Join Parallelism

- sometimes the amount of work to parallelize is not known upfront
- fork/join allows one to perform work on other threads ("fork"), and then to wait until these tasks are finished ("join")
- often recursive parallelism structure

# Naive Merge Sort with Fork/Join (TBB)

```
const ptrdiff_t limit = 1024;

template<class Iter>
void merge_sort(Iter first, Iter last) {
    if (last - first > limit) {
        Iter middle = first + (last - first) / 2;
        tbb::task_group g; // alternative: tbb::parallel_invoke
        g.run([&]{ merge_sort(first, middle); } );
        merge_sort(middle, last);
        g.wait();
        std::inplace_merge(first, middle, last);
    } else {
        merge_sort_serial(first, last);
    }
}
```

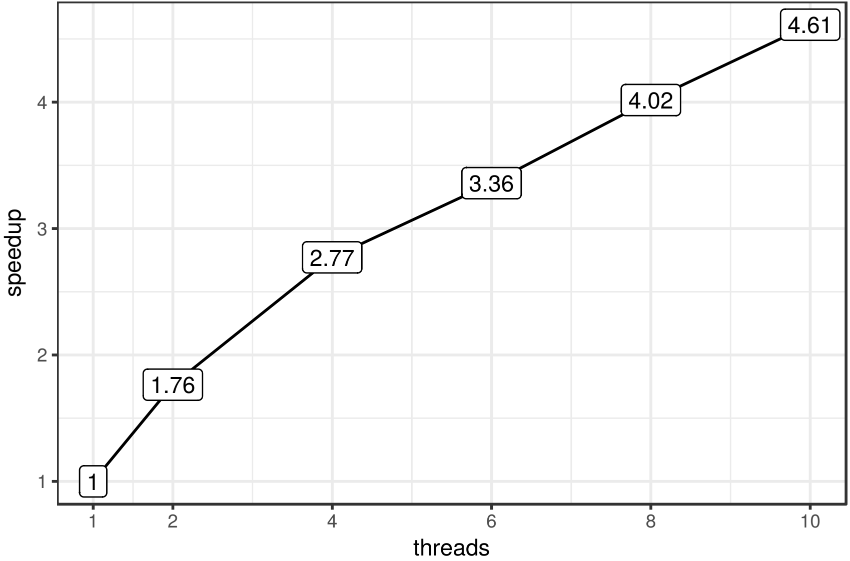
# Analysis

- What is the speedup for sorting $n$ elements with infinite cores?
- serial execution: $\log_2(n) \cdot n$
- rough upper bound (using Amdahl's law):
  - the final merge is serial: $n$
  - lower bound for fraction of serial part $\frac{n}{\log_2(n) \cdot n} = \frac{1}{\log_2(n)}$
  - using Amdahl's law the maximum speedup is $\frac{1}{\frac{1}{\log_2(n)}} = \log_2(n)$
  - for example, if $n = 2^{20}$ the upper bound is $\log_2(n) = 20$
- tighter upper bound:
  - parallel execution: $\sum_{i=0}^{\log_2(n)-1} \frac{n}{2^i} = n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$
  - for example, if $n = 2^{20}$ the upper bound is

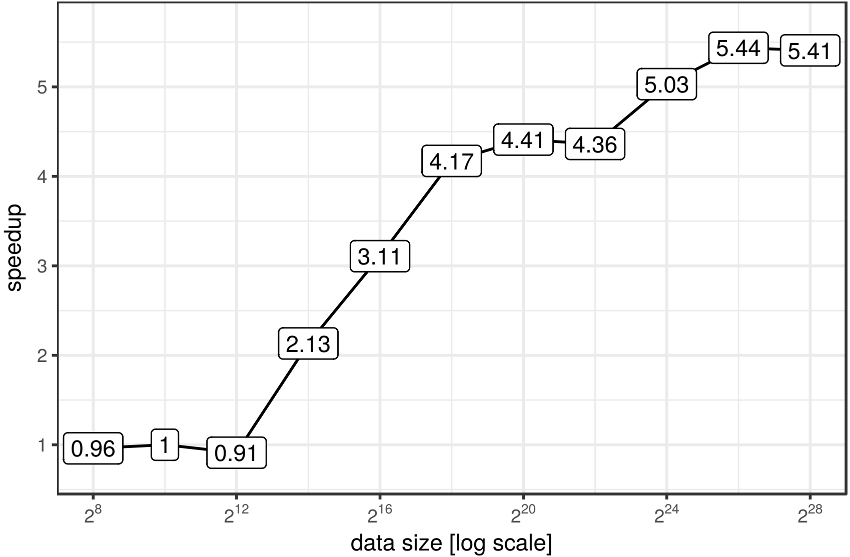
$$\frac{20n}{2n} = 10$$

- (both analyses assume that each level recursion level takes the same amount of time)

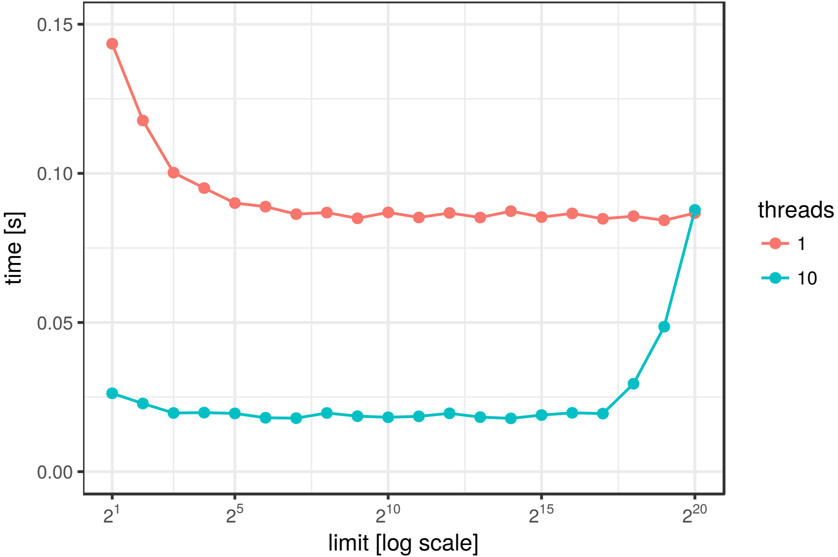
# Speedup, $n = 2^{20}$



# Speedup with 10 Threads



# Parallelization Overhead, $n = 2^{20}$



# Parallel Merge (1)

```
template<typename It>
void parallelMerge(It begin1, It end1, It begin2, It end2, It out) {
   tbb::parallel_for(ParallelMergeRange<It>(begin1, end1, begin2, end2, out),
      [&](ParallelMergeRange<It>& r) {
         std::merge(r.begin1, r.end1, r.begin2, r.end2, r.out); },
      tbb::simple_partitioner());
}

template<typename It>
struct ParallelMergeRange { // TBB range concept
   It begin1, end1, begin2, end2, out;

   bool empty() const { return (end1-begin1) + (end2-begin2)==0; }

   bool is_divisible() const {
     return std::min(end1-begin1, end2-begin2) > limit; }

   ParallelMergeRange(It begin1_, It end1_, It begin2_, It end2_, It out_) :
      begin1(begin1_), end1(end1_), begin2(begin2_), end2(end2_), out(out_) {}
```

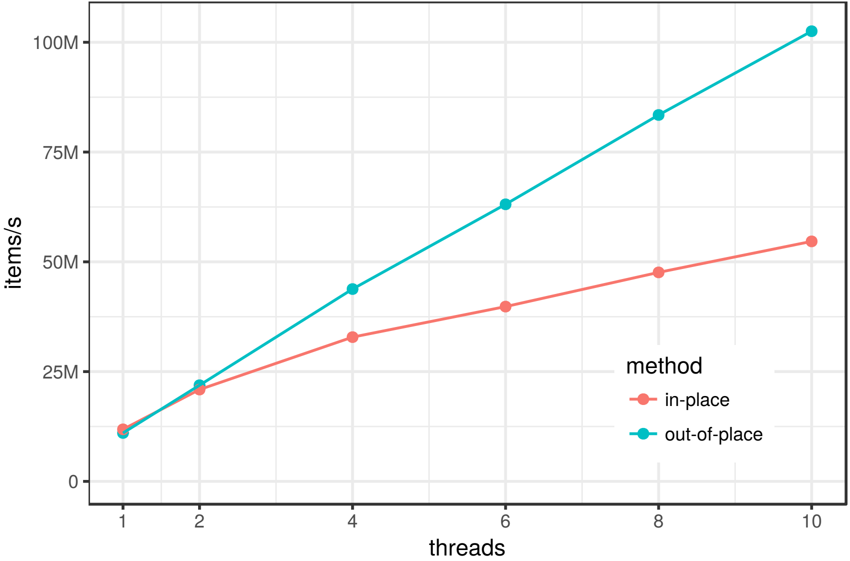
# Parallel Merge (2)

```
    // Splitting constructor (splits r into two subranges)
    ParallelMergeRange(ParallelMergeRange& r, tbb::split) {
        if ((r.end1-r.begin1) < (r.end2-r.begin2)) {
            // first range should be the larger one
            std::swap(r.begin1, r.begin2);
            std::swap(r.end1, r.end2);
        }
        It m1 = r.begin1 + (r.end1-r.begin1)/2;
        It m2 = std::lower_bound(r.begin2, r.end2, *m1);
        begin1 = m1;
        begin2 = m2;
        end1 = r.end1;
        end2 = r.end2;
        out = r.out + (m1-r.begin1) + (m2-r.begin2);
        r.end1 = m1;
        r.end2 = m2;
    }
}; // struct ParallelMergeRange
```

# Parallel Out-Of-Place Merge Sort

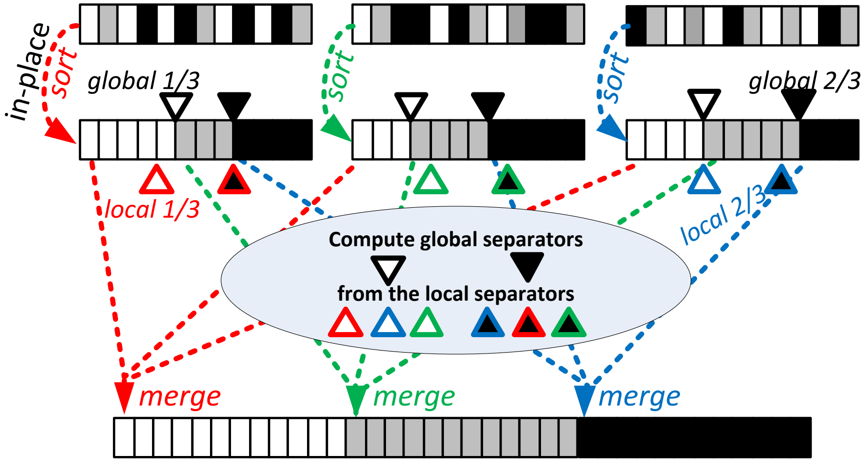
```
template<class It>
void parallelMergeSort(It first, It last, It out, bool inplace=false) {
   if ((last-first) < limit) {
      merge_sort_serial(first, last);
      if (!inplace)
         std::move(first, last, out);
   } else {
      It mid = first + (last-first)/2;
      It outMid = out + (mid-first);
      It outLast = out + (last-first);
      tbb::parallel_invoke(
         [&]() { parallelMergeSort(first, mid, out, !inplace); },
         [&]() { parallelMergeSort(mid, last, outMid, !inplace); });
      if (inplace)
         parallelMerge(out, outMid, outMid, outLast, first);
      else
         parallelMerge(first, mid, mid, last, out);
   }
}
```

# Scalability, $n = 2^{20}$



# HyPer's Parallel Merge Sort

1. divide input data statically, each thread sorts its fraction
2. determine separators, compute output positions (prefix sums)
3. merge into output array

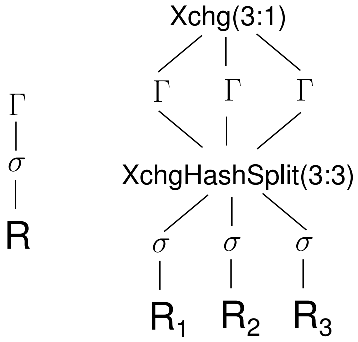


# Pitfalls in Parallel Code

- non-scalable algorithm
  - re-think algorithm
- load imbalance
  - break work into smaller tasks, dynamically schedule these between threads
- task overhead: managing tasks takes more time than the actual work
  - set a minimum per-thread tasks size (not too small, not to large)

# Volcano-Style Parallelism

- *plan-driven* approach:
  - optimizer statically determines at query compile time how many threads should run
  - instantiates one query operator plan for each thread
  - connects these with exchange operators, which encapsulate parallelism and manage threads
- Elegant model which is used by many systems

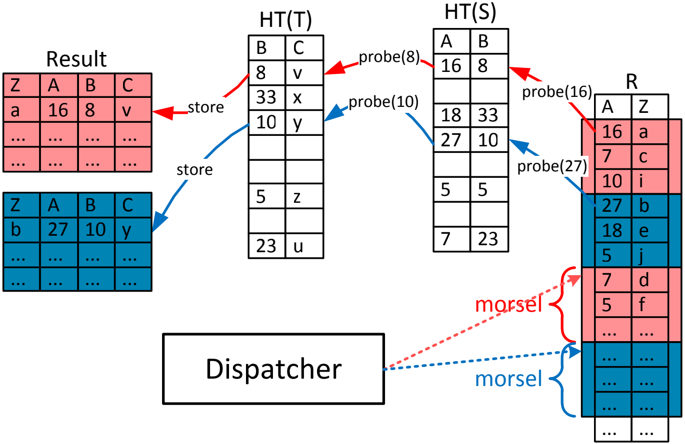


# Volcano-Style Parallelism (2)

+ operators are largely oblivious to parallelism
– static work partitioning can cause load imbalances
– degree of parallelism cannot easily be changed mid-query
– overhead:
  - thread oversubscription causes context switching
  - hash re-partitioning often does not pay off
  - exchange operators create additional copies of the tuples

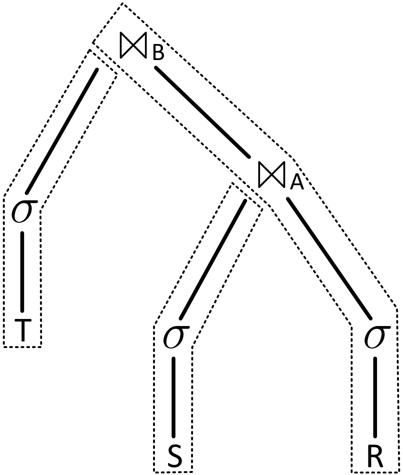
# Morsel-Driven Query Execution (1)

- break input into constant-sized work units (“morsels”)
- dispatcher assigns morsels to worker threads
- # worker threads = # hardware threads
- operators are designed for parallel execution



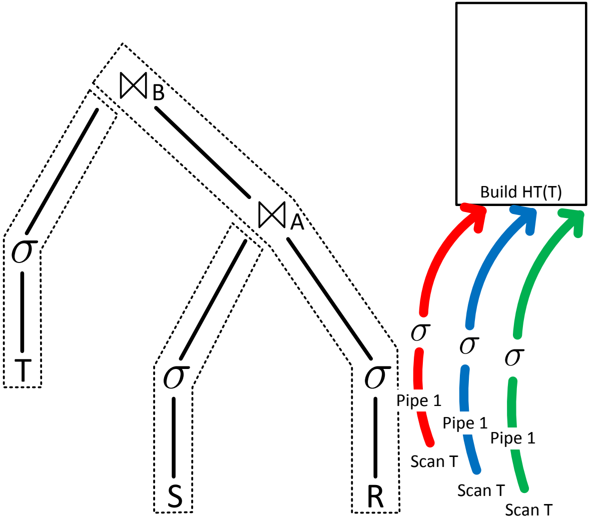
# Pipeplines

- each pipeline is parallelized individually using all threads



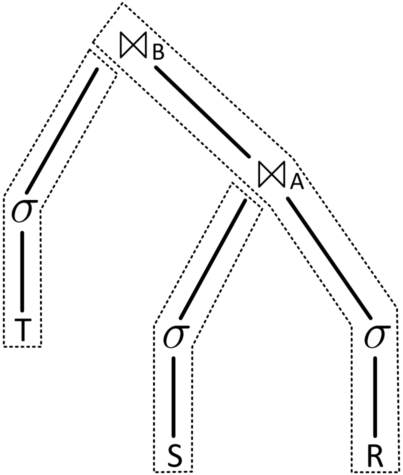
# Pipeplines

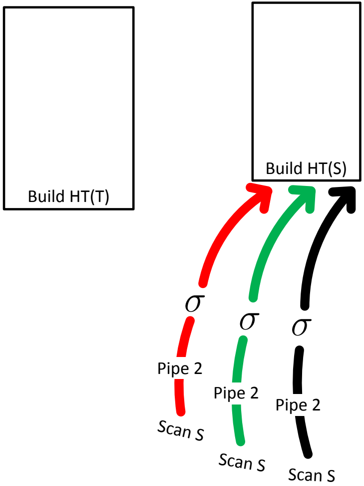
- each pipeline is parallelized individually using all threads



# Pipeplines

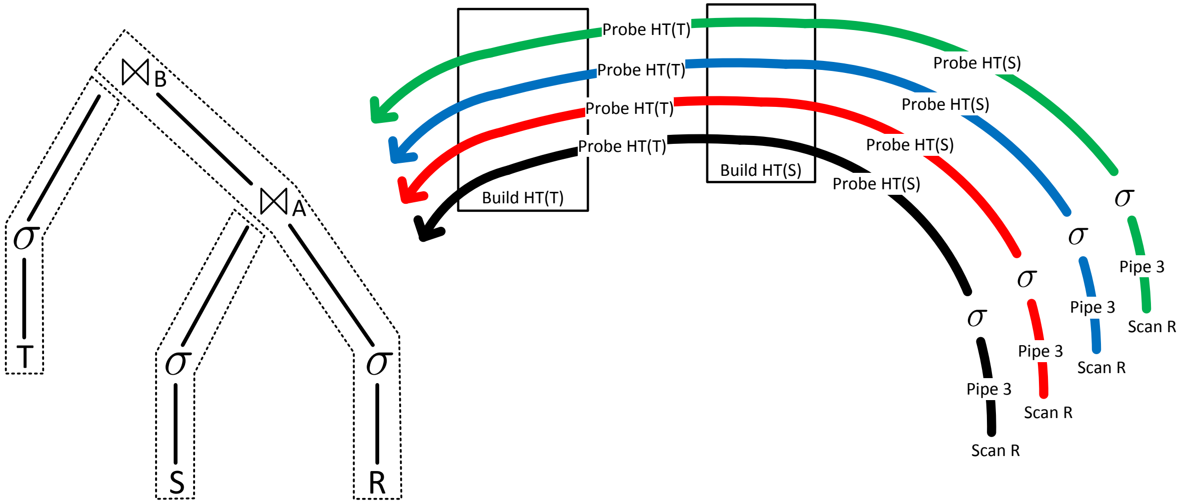
- each pipeline is parallelized individually using all threads



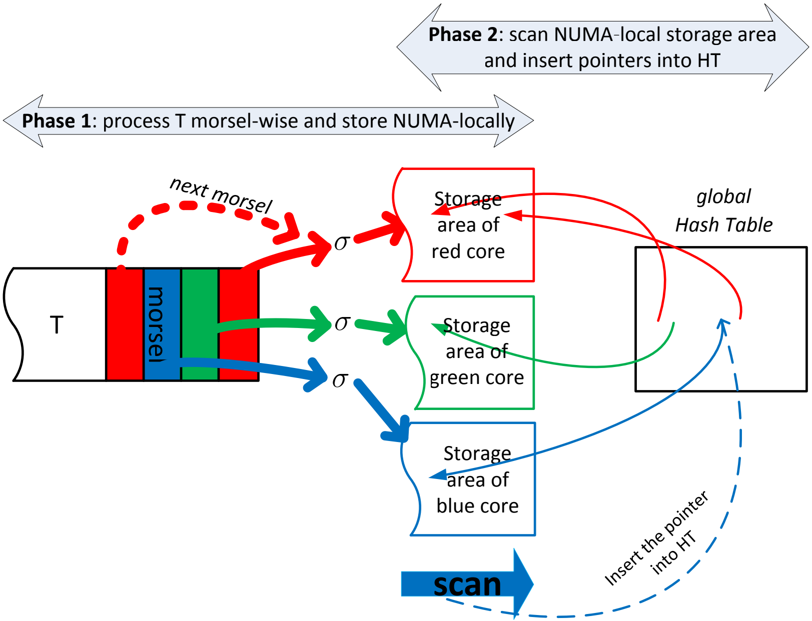



# Pipeplines

- each pipeline is parallelized individually using all threads

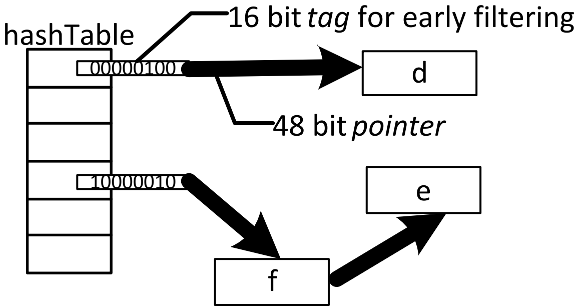


# Parallel Hash Table Construction



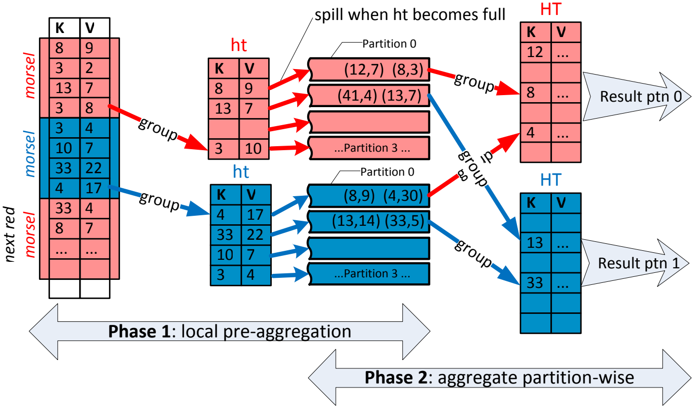
# Hash Tagging

- unused bits in pointers act as a cheap bloom filter



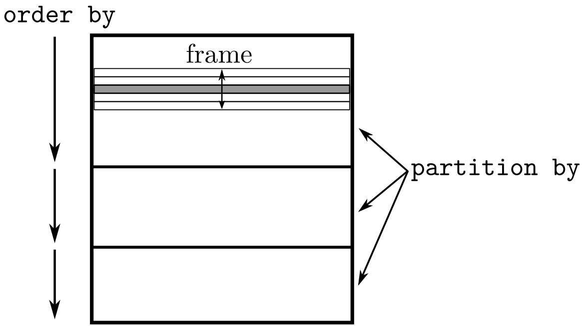
# Aggregation/Group By

- parallel aggregation is one of the most difficult relational operators
- main challenge: behaves very differently depending on whether there are few or many distinct keys

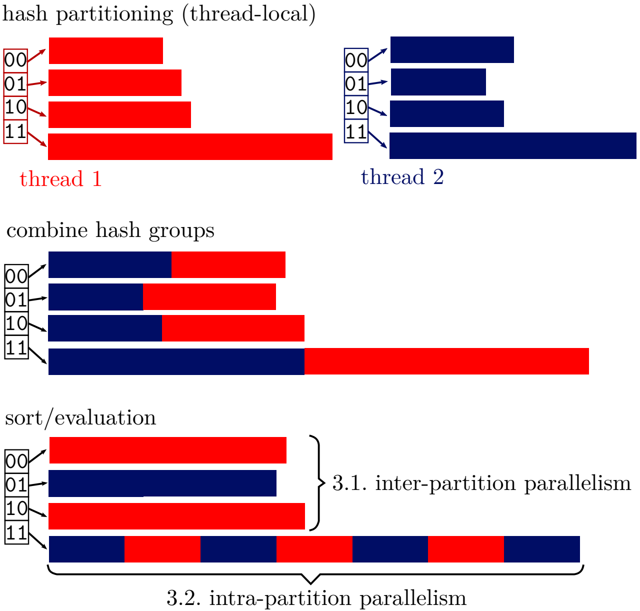


# Window Functions: Semantics

```
select location, time, avg(value) over
  (partition by location
   order by time
   range between interval '2' day preceding
                 and interval '1' day following)
from measurement
```



# Window Functions: Semantics



# References


- *Structured Parallel Programming: Patterns for Efficient Computation*, McCool and Robison and Reinders, Morgan Kaufmann, 2012
- *Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age*, Leis and Boncz and Kemper and Neumann, SIGMOD 2014
- *Encapsulation of Parallelism in the Volcano Query Processing System*, Graefe, SIGMOD 1990