Distributed Indexing

Indexing, session 8

Slides by: Jesse Anderton

Distributing Indexing

The scale of web indexing makes it infeasible to maintain an index on a single computer. Instead, we distribute the task across a cluster (or more).

The traditional way to provision a data center is to buy several large mainframes running a massive database, such as Oracle. In contrast, distributed indexes generally run on large numbers of cheap computers that are expected to fail and be replaced frequently.

A primary tool for running software across these clusters is MapReduce, and similar frameworks.

By Analogy

Suppose you have a very large file of credit card transactions. Each line has a credit card number and a transaction amount. You wish to know the total charged to each card.

You could use a hash table in memory, but if there are enough numbers you will run out of space.

If the file was sorted, you could just count amounts in a single pass.

Similarly, MapReduce programs depend on proper sorting to group sub-tasks together on a single computer.

Credit Card Log

```
4404-5414-0324-3881  $78.62

4532-7096-2202-7659  $26.92

4787-8099-6978-7089  $451.05

4485-0342-4391-4731  $5.23

4916-2026-7936-6663  $34.50
```

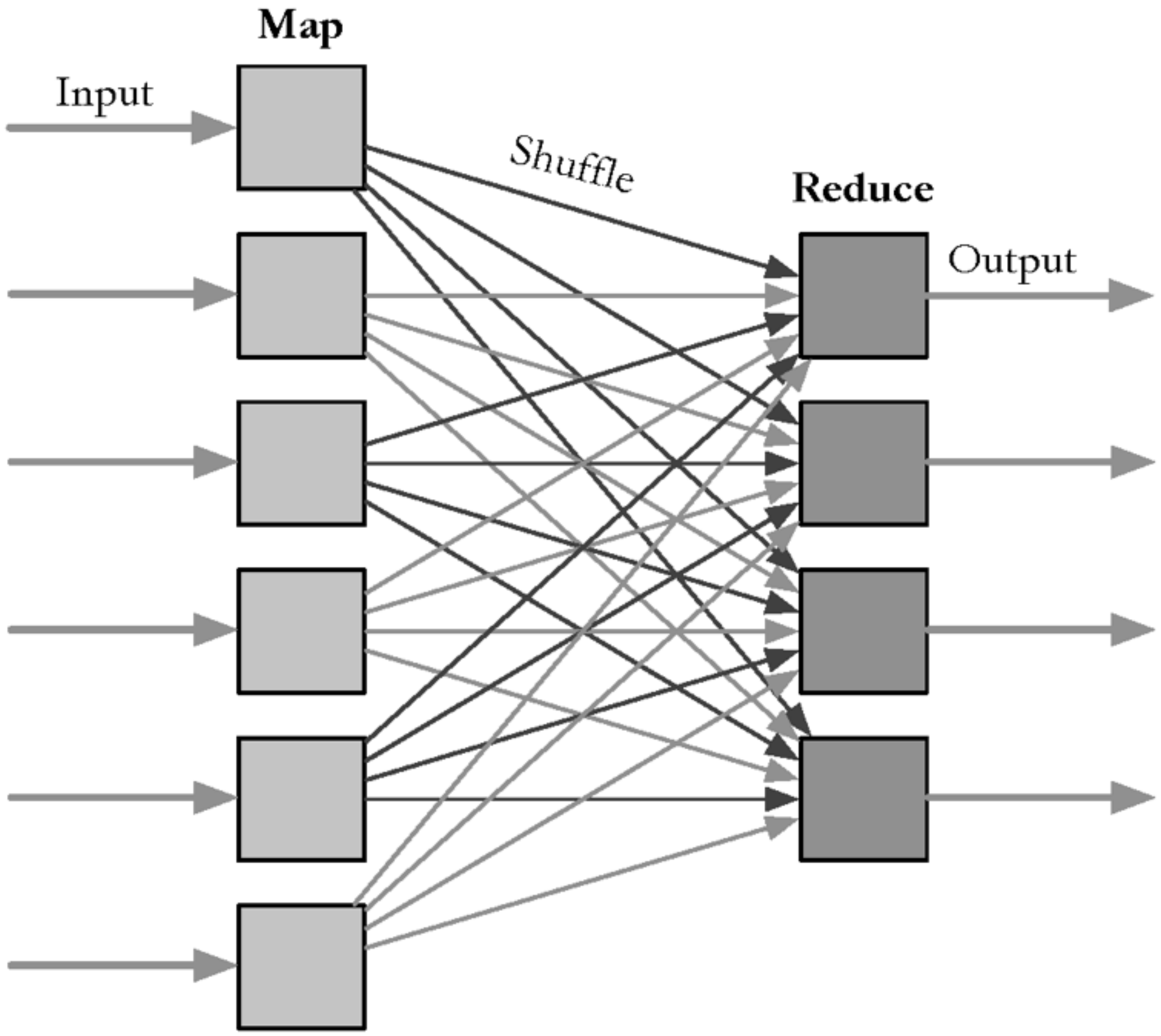
MapReduce

MapReduce is a distributed programming framework focused on data placement and distribution.

Mappers take a list of input records and transform them, generally into a list of the same length.

Reducers take a list of input records and transform them, generally into a single value.

A chain of mappers and reducers is constructed to transform a large dataset into a (usually simpler) output value.



MapReduce

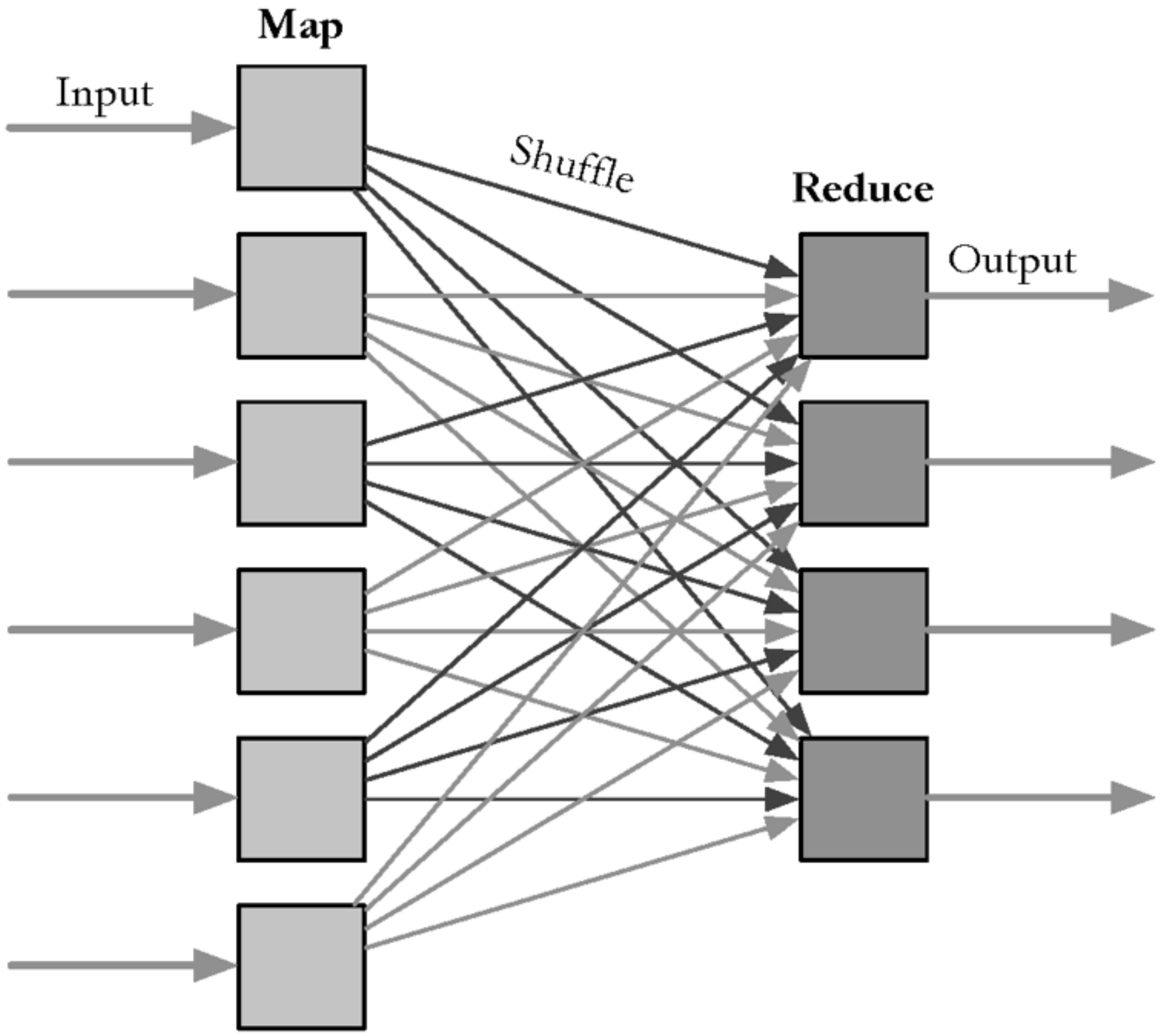
Basic Process:

1. The raw input is sent to the mappers, which transform it into a sequence of `<key, value>` pairs.

2. Shufflers take the mapper output and sent it to the reducers. A given reducer typically gets all the pairs with the same `key`.

3. Reducers process batches of all pairs with the same `key`.

The Mapper and Reducer jobs must be **idempotent**, meaning that they deterministically produce the same output from the same input. This provides fault tolerance, should a machine fail.



Example: Credit Cards

```
procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        card ← record.card
        amount ← record.amount
        Emit(card, amount)
    end while
end procedure

procedure ReduceCreditCards(key, values)
    total ← 0
    card ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit(card, total)
end procedure
```

This mapper and reducer will count the number of distinct credit card numbers in the input.

The mapper **emits** (outputs) pairs whose keys are credit card numbers.

The reducer processes a batch of pairs with the same credit card number, and emits the total for the card.

Example: Indexing

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, number:position)
            position = position + 1
        end for
    end while
end procedure

procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```

This mapper and reducer index a collection of documents.

The mapper emits pairs whose keys are terms and whose values are `docid:position` pairs.

The reducer encodes all postings for the same term.

How can `WriteWord()` and `EncodePosting()` be written to have idempotence?

Map Reduce Summary

MapReduce is a powerful framework which has been extended in many interesting ways to support sophisticated distributed algorithms.

Here, we've seen a simple approach to indexing based on MapReduce. Consider how we might process queries with MapReduce.

Next, we'll take a look at a distributed storage system to complement our distributed processing.

Big Table

Storage systems such as BigTable are natural fits for distributed algorithm execution.

Google invented BigTable to handle its index, document cache, and most of its other massive storage needs.

This has produced a whole generation of distributed storage systems, called NoSQL systems. Some examples include MongoDB, Couchbase, etc.

Distributed Storage

BigTable was developed by Google to manage their storage needs.

It is a distributed storage system designed to scale across hundreds of thousands of machines, and to gracefully continue service as machines fail and are replaced.

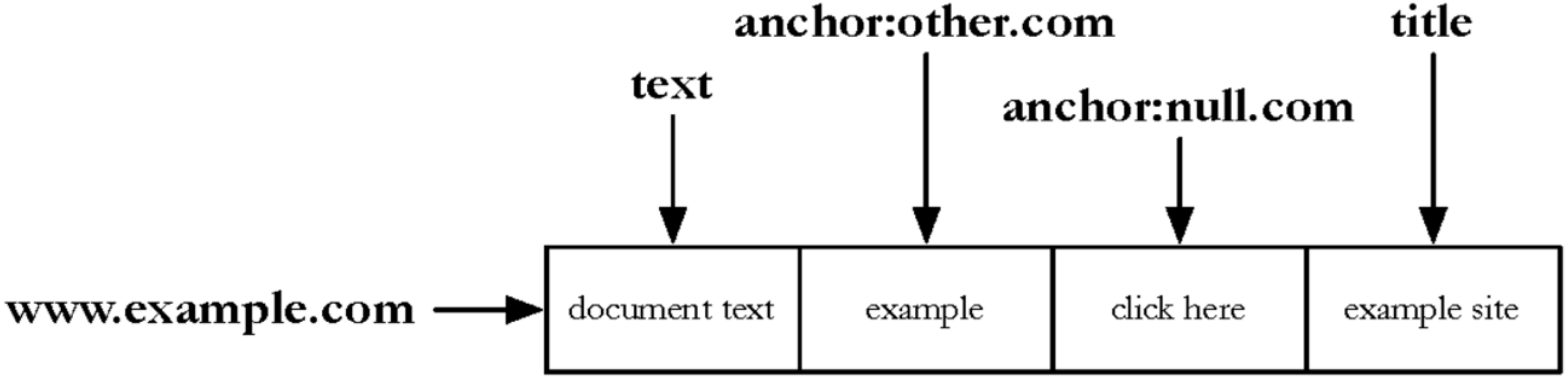
Storage systems such as BigTable are natural fits for processes distributed with MapReduce.

"A Bigtable is a sparse, distributed, persistent multidimensional sorted map." –Chang et al, 2006.

BigTable Rows

The data in BigTable is logically organized into rows. For instance, the inverted list for a term can be stored in a single row.

A single cell is identified by its row key, column, and timestamp. Efficient methods exist for fetching or updating particular groups of cells. Only populated cells consume filesystem space: the storage is inherently sparse.

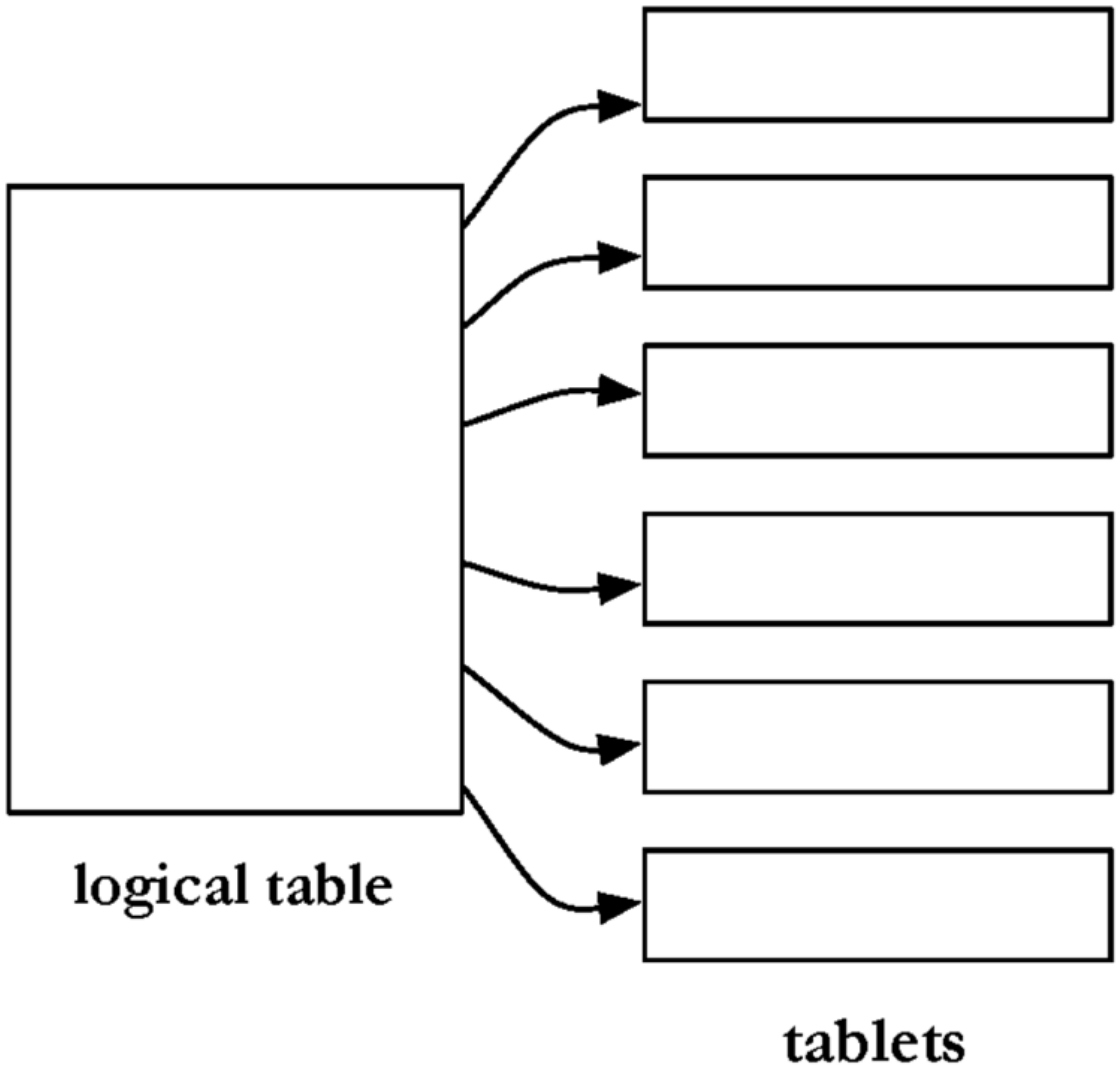


BigTable Tablets

BigTable rows reside within logical tables, which have pre-defined columns and group records of a particular type.

The rows are subdivided into ~200MB tablets, which are the fundamental underlying filesystem blocks. Tablets and transaction logs are replicated to several machines in case of failure.

If a machine fails, another server can immediately read the tablet data and transaction log with virtually no downtime.



BigTable Operations

All operations on a BigTable are row-based operations.

Most SQL operations are impossible here: no joins or other structured queries.

BigTable rows can have massive numbers of columns, and individual cells can contain large amounts of data. For instance, it’s no problem to store a translation of a document into many languages, each in its own column of the same row.

Query Processing

Both doc-at-a-time and term-at-a-time have their advantages.

- Doc-at-a-time always knows the best k documents, so uses less memory.
- Term-at-a-time only reads one inverted list at a time, so is more disk efficient and more easily parallelized (e.g., use one cluster node per query term).

Query Processing

There are two main approaches to scoring documents for a query on an inverted index.

- **Document-at-a-time** processes all the terms' posting lists in parallel, calculating the score for each document as it's encountered.

- **Term-at-a-time** processes posting lists one at a time, updating the scores for the documents for each new query term.

There are optimization strategies for either approach that significantly reduce query processing time.

Doc-at-a-Time Processing

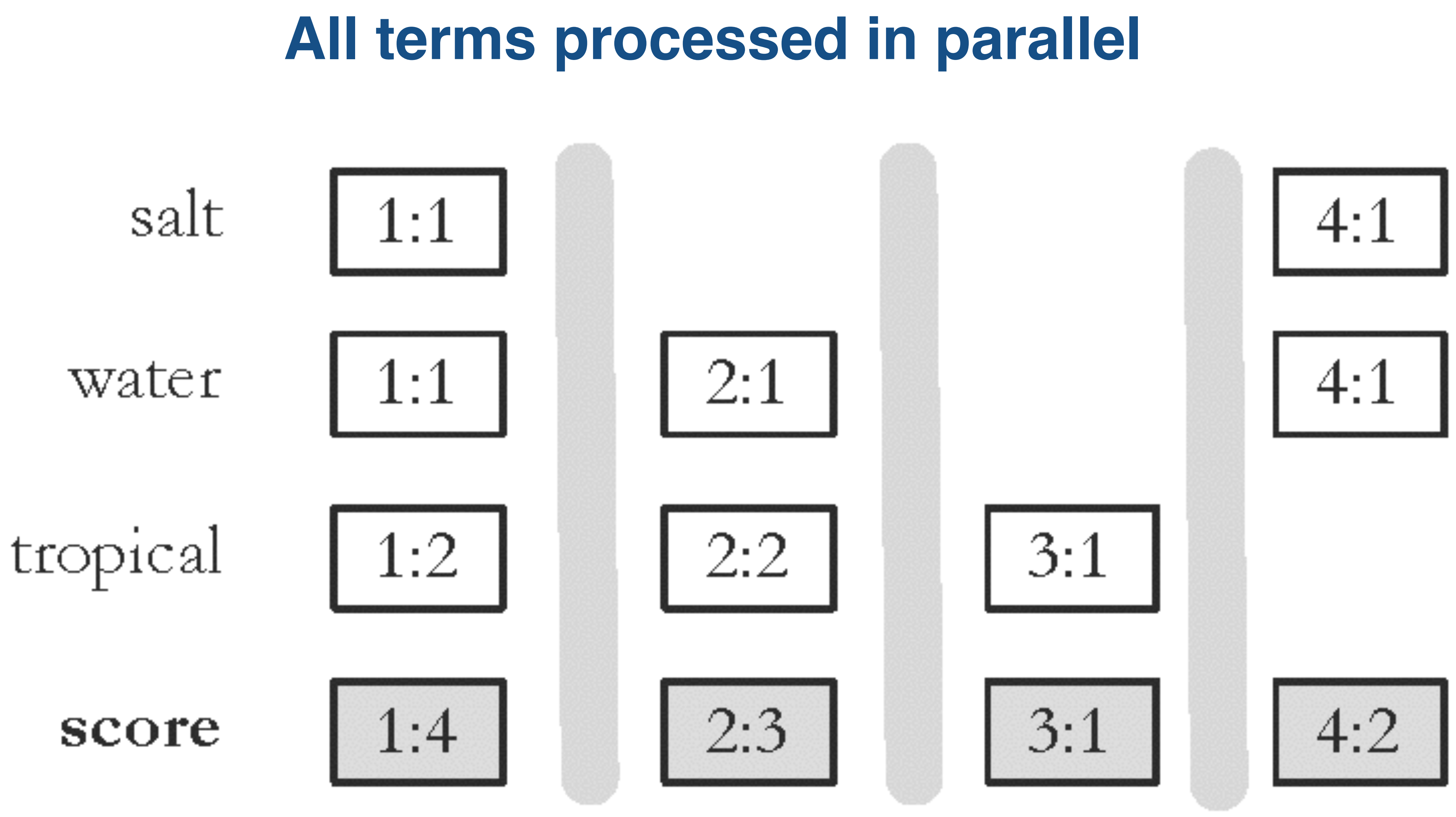
We scan through the postings for all terms simultaneously, calculating the score for each document.

We remember scores for the top k documents found so far.

Recall that the document score has the form:

$$\sum_{w \in q} f(w) \cdot g(w)$$

for document features $f(w)$ and query features $g(w)$.



Doc-at-a-Time Algorithm

Get the top k documents for query Q from index I, with doc features f and query features g

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)        ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

This algorithm implements doc-at-a-time retrieval.

It uses a list L of inverted lists for the query terms, and processes each document in sequence until all have been scored.

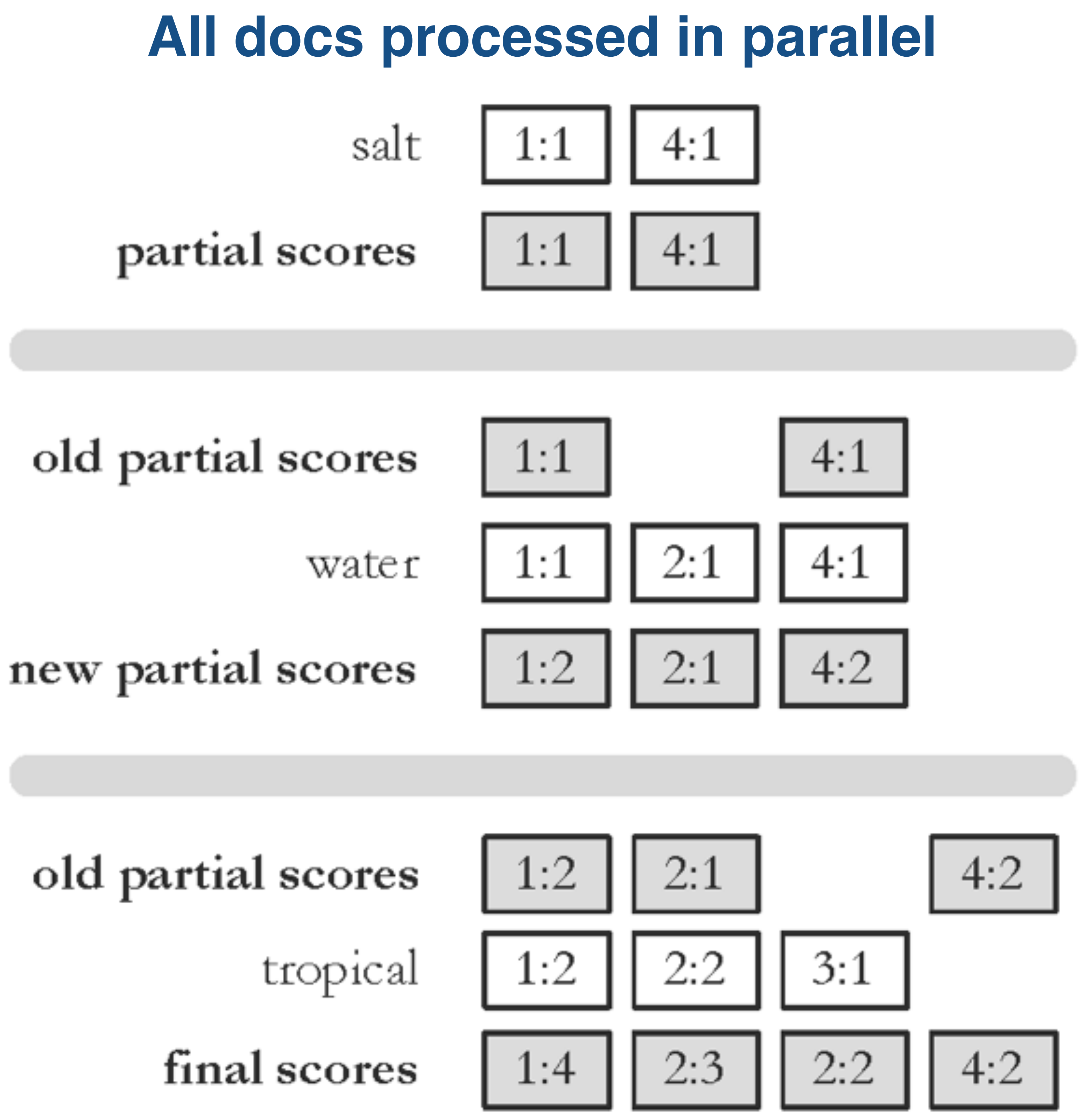
The documents are placed into the priority queue R so the top k can be returned.

Term-at-a-Time Processing

For term-at-a-time processing, we read one inverted list at a time.

We maintain partial scores for the documents we've seen so far, and update them for each term.

This may involve remembering more document scores, because we don't necessarily know which documents will be in the top k (but sometimes we can guess).



Term-at-a-Time Algorithm

Get the top k documents for query Q from index I, with doc features f and query features g

procedure TermAtATimeRetrieval(Q, I, f, g k)
    $A \leftarrow$ HashTable()
    $L \leftarrow$ Array()
    $R \leftarrow$ PriorityQueue(k)
    for all terms w_i in Q **do**
        $l_i \leftarrow$ InvertedList(w_i, I)
        L.add(l_i)
    end for
    for all lists $l_i \in L$ **do**
        while l_i is not finished **do**
            $d \leftarrow l_i$.getCurrentDocument()
            $A_d \leftarrow A_d + g_i(Q)f(l_i)$
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A **do**
        $s_d \leftarrow A_d$ ▷ Accumulator contains the document score
        R.add(s_d, d)
    end for
    return the top k results from R
end procedure

This algorithm implements term-at-a-time retrieval.

It uses an accumulator A of partial document scores, and updates a document's score when the doc is encountered in an inverted list.

Once all scores are calculated, we place the documents into a priority queue R so the top k can be returned.

Optimized Query Processing

There are many more ways to speed up query processing. Rapid query responses are essential for the user experience of search engines, so this is a heavily studied area.

In general, methods can be categorized as *safe methods*, which always return the top k documents, or *unsafe methods* which just return k "pretty good" documents.

Next, we'll look at ways we can arrange indexes to speed up results for common or easy queries.

Optimization Strategy

There are two main approaches to query optimization:

1. Read less data from the inverted lists
   e.g., use *skip lists* to jump past "unpromising" documents

2. Calculate scores for fewer documents
   e.g., use *conjunctive processing*: require documents to have all query terms

Conjunctive Doc-at-a-Time

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    d ← −1
    while all lists in L are not finished do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() > d then
                d ← l_i.getCurrentDocument()
            end if
        end for
        for all inverted lists l_i in L do
            l_i.skipForwardToDocument(d)
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q)f_i(l_i)            ▷ Update the document score
                l_i.movePastDocument( d )
            else
                d ← −1
                break
            end if
        end for
        if d > −1 then R.add( s_d, d )
        end if
    end while
    return the top k results from R
end procedure
```

This doc-at-a-time implementation only considers documents which contain all query terms.

Note that we assume that docids are encountered in sorted order in the inverted lists.

Conjunctive Term-at-a-Time

```
procedure TermAtATimeRetrieval(Q, I, f, g, k)
    A ← Map()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        d_0 ← −1
        while l_i is not finished do
            if i = 0 then
                d ← l_i.getCurrentDocument()
                A_d ← A_d + g_i(Q)f(l_i)
                l_i.moveToNextDocument()
            else
                d ← l_i.getCurrentDocument()
                d′ ← A.getNextAccumulator(d)
                A.removeAccumulatorsBetween(d_0, d′)
                if d = d′ then
                    A_d ← A_d + g_i(Q)f(l_i)
                    l_i.moveToNextDocument()
                else
                    l_i.skipForwardToDocument(d′)
                end if
                d_0 ← d′
            end if
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d                    ▷ Accumulator contains the document score
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

This is the term-at-a-time version of conjunctive processing.

Here, we delete accumulators for documents which are missing query terms.

Threshold Methods

If we only plan to show the user the top k documents, that implies that all documents we return have scores at least as good as the k^{th}-best document.

Let τ be the minimum score of any document we return. We can use an estimate of τ to stop processing low-scoring documents early.

- For doc-at-a-time, our estimate τ' is the score of the k^{th}-best doc seen so far
- For term-at-a-time, τ' is the k^{th}-largest score in any accumulator

Example: Threshold Filtering

Return the top two documents. All scores are between 0 and 1. We score documents by taking the dot product of document and query scores.

Query term vector: [0.7, 0.1, 0.2]

Doc 1: [0.3, 0.4, 0.5] **Score:** $0.3 \times 0.7 + 0.4 \times 0.1 + 0.5 \times 0.2 = 0.35$

Doc 2: [0.5, 0.1, 0.1] **Score:** $0.5 \times 0.7 + 0.1 \times 0.1 + 0.1 \times 0.2 = 0.38$

Doc 3: [0.01, 1, 1] **Score:** $0.01 \times 0.7 + 1 \times 0.1 + 1 \times 0.2 = 0.307$

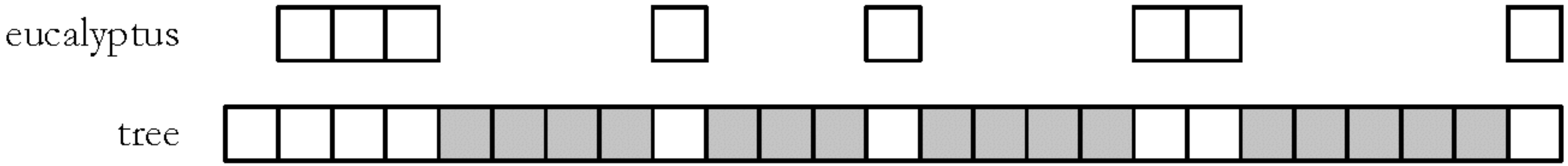
For doc 3, even though the last two terms have perfect scores the document was rejected. We can tell from the first term that it will never score highly enough to be retrieved. We don't even have to look at the second or third terms.

MaxScore Method

The **MaxScore Method** is an algorithm for efficiently retrieving the top k documents by comparing the top score a document could have to the estimate τ'.

At index time, we compute the largest score μ_w any document achieved for each term w. We use these scores at query time to estimate the maximum score any document could have, based on the information so far.

For instance, suppose $\tau' > \mu_{tree}$ in the below lists for the query “eucalyptus tree.” We can skip all the grey documents, because no score for tree is enough to be included without also matching eucalyptus.



Unsafe Optimizations

There are also many unsafe optimizations we could use. These may not return the top k documents, but they will generally return k "good enough" documents.

- Query processing can be abandoned early, e.g., after some elapsed time or minimum document score is reached.

- High-frequency terms can be ignored in term-at-a-time queries, and documents at the end of the lists can be ignored in doc-at-a-time.

When we plan to process partial postings, it's a good idea to sort them by some sort of quality score (e.g., PageRank) so we will probably return high-quality documents.

Tiered Indexes

The organization of indexes in a large-scale search engine is important for rapid query processing.

Inverted lists can be sorted in various ways to improve inexact top k retrieval performance, and tiered indexes are often used to handle “easy” queries quickly while still offering good performance for rarer, more difficult queries.

Good multi-level caching strategies are also essential for achieving good performance, particularly for web and peer-to-peer search.

Champion Lists

Champion Lists are inverted lists for terms which contain only the highest-scoring documents for that term.

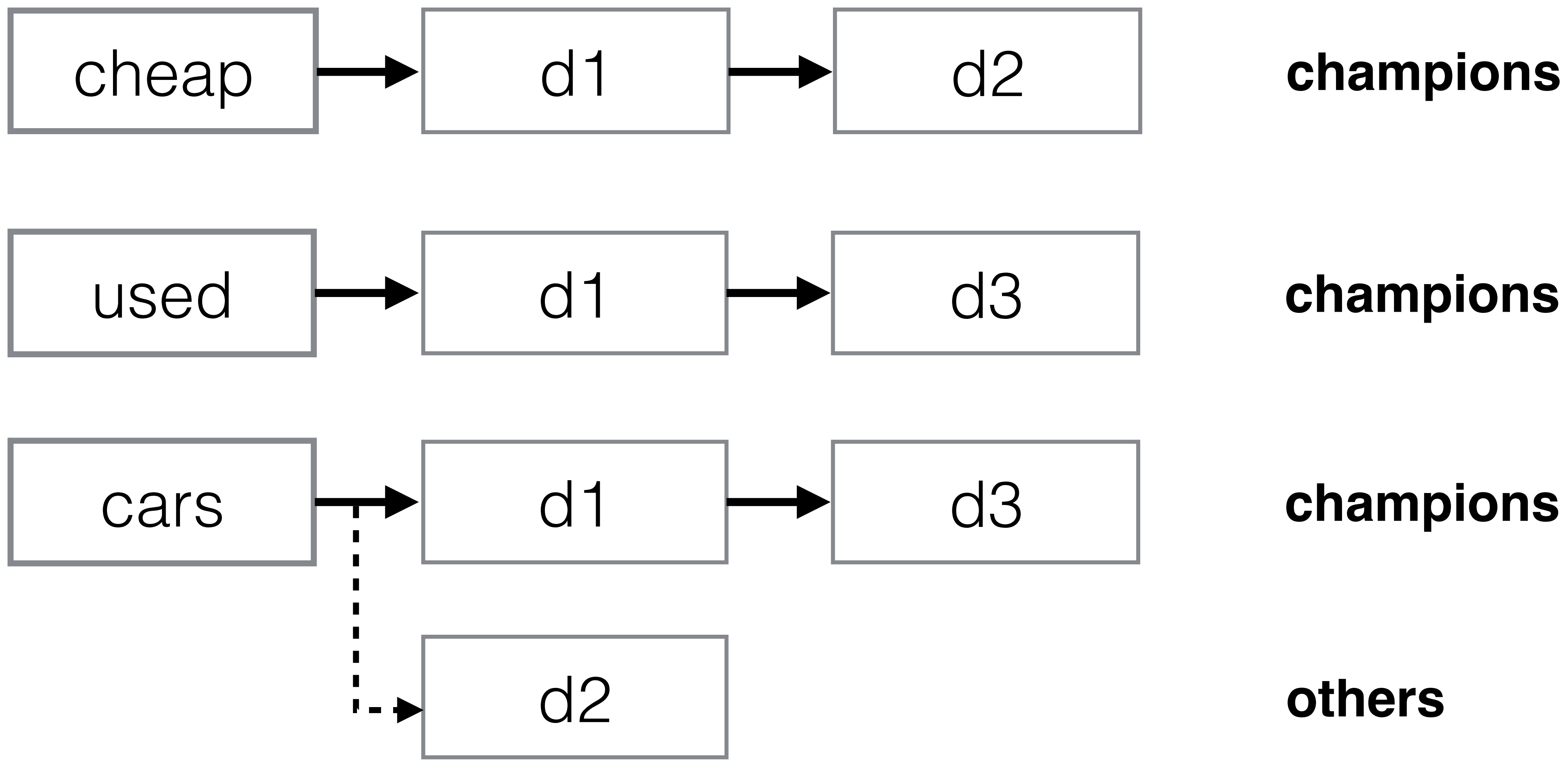
At indexing time, we compute a document's matching score for a term. If it's one of the top r documents, we add it to the champion list.

At query time, we first match documents in the champion list for any query term, and only proceed to other documents if that didn't find enough results.

We can pick larger r for terms with higher df. Why would this help?

Champion Lists

	d1	d2	d3
tf	2	6	0
tf	1	0	6
tf	8	3	5



Sorting by Quality

As a generalization of champion lists, we can sort the postings for a term by some document quality score q_d. Suppose the quality score is part of our matching function:

$$score(D, Q) = \lambda q_D + (1 - \lambda) \sum_{w \in Q} f(w) \cdot g(w)$$

Recall that we want to sort the postings by a common value so we can easily merge them. We previously sorted by docid.

Sorting by global document quality still allows efficient merging, though sorting by a term-based matching score would not.

Postings sorted by quality

	d1	d2	d3
q	0.5	0.25	0.75

cheap → d1 → d2

used → d3 → d1

cars → d3 → d1 → d2

Impact Ordering

If we use term-at-a-time processing, we can sort the lists in different orders.

Impact Ordering sorts lists by some notion of term relevance. As a simple example, $tf_{w,d}$ can be used.

Here, we often stop processing documents early in each list. We may process query terms in order of decreasing df, and stop processing each list when document scores stop changing much. We may also skip low-df terms.

Postings sorted by tf

	d1	d2	d3
tf	2	6	0
tf	1	0	6
tf	8	3	5

cheap → d2 → d1

used → d3 → d1

cars → d1 → d3 → d2

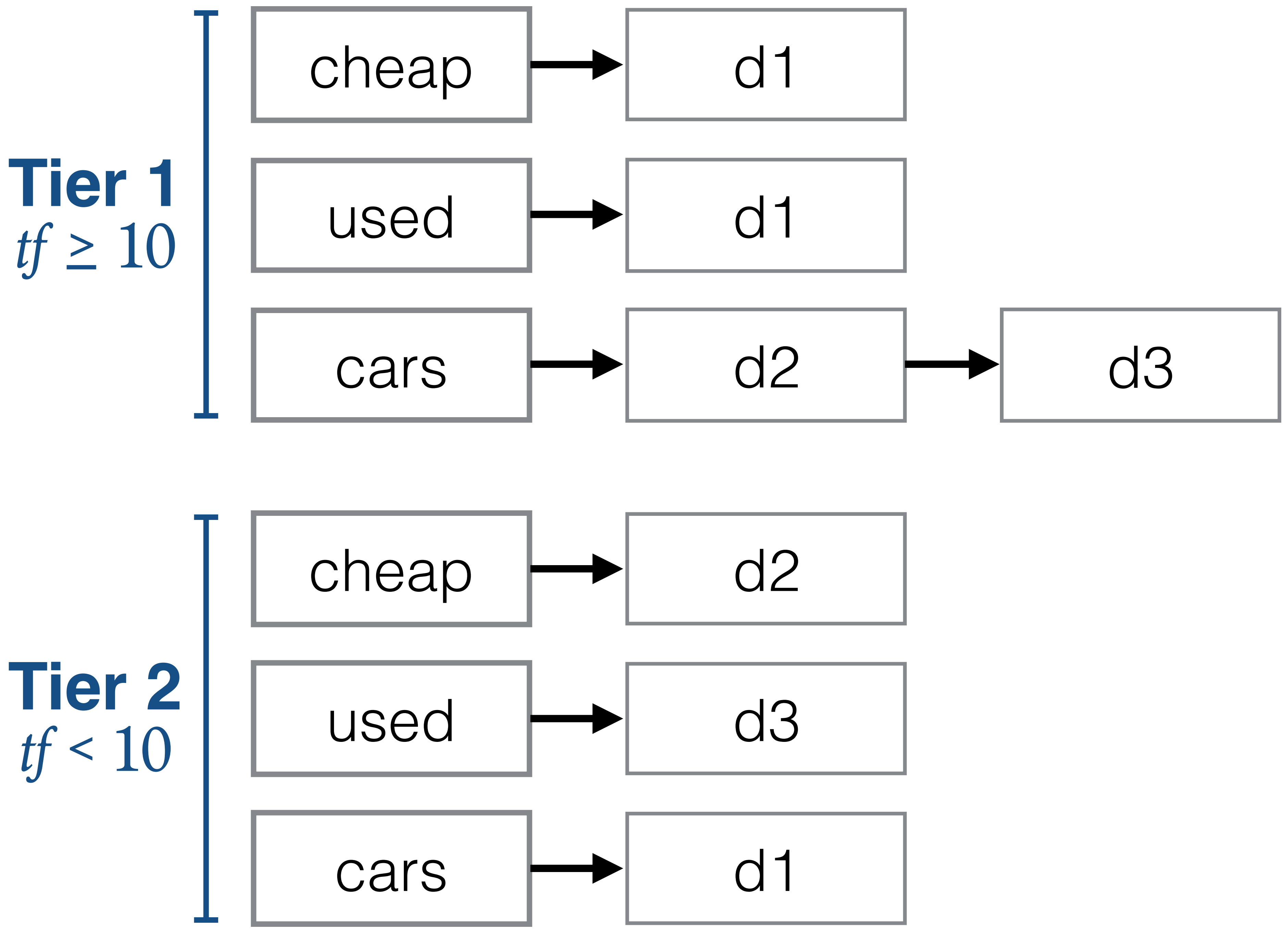
Tiered Indexes

Tiered Indexes take these ideas further. We use multiple indexes. Documents likely to have the highest scores are in the first index, and subsequent indexes have progressively worse documents.

We process queries in one index at a time, stopping when we find enough documents. Only a few queries will need all indexes.

Early tiers are often optimized for speed. For instance, the top tier might be held in RAM, while lower tiers are on disk.

	d1	d2	d3
tf	27	3	0
tf	17	0	6
tf	8	13	16



Query Caching

Caching also plays an essential role in improving query performance for large search engines. Many forms of caching are used.

- Results for common queries are cached. A substantial fraction of queries are run by many users (e.g., “facebook”).
- Merged inverted lists for common sets of query terms are cached. This is particularly useful for common phrases (e.g., “new york city”).
- Caching is particularly important in Peer-to-peer search, where a query may download cached results from other peers.

Caching is often implemented in a multi-level way, e.g., the query cache is checked first, then a cache of merged lists is checked, and finally a cache of individual inverted lists.

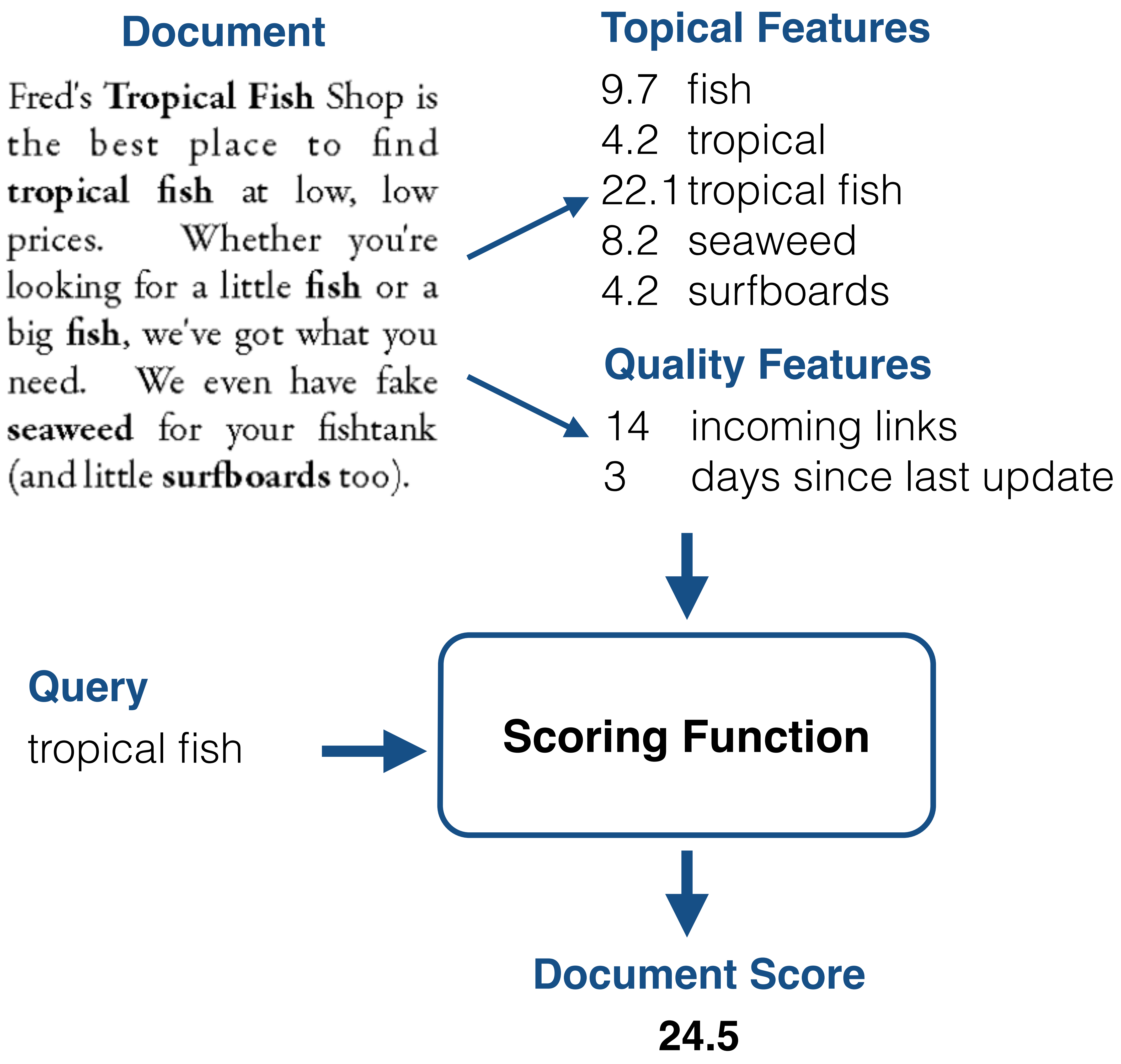
Indexing: Wrap Up

Inverted Indexes

Inverted indexes are data structures meant to enable rapid query processing.

We store many types of information in indexes; modern scoring functions combine evidence from many topical and quality features.

The indexing process needs to be carefully engineered to create and update inverted lists efficiently, taking data volume into account. In particular, good index compression is key.



Query Processing

Queries may be processed in doc-at-a-time or term-at-a-time order; either approach has its advantages and optimization strategies.

Indexes are often sorted, tiered, and cached in order to support rapid results for common or easy queries and good results for uncommon or difficult queries.