WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web*

Roy Goldman, Jennifer Widom

Stanford University
{royg,widom}@cs.stanford.edu
www-db.stanford.edu

Abstract

We present WSQ/DSQ (pronounced "wisk-disk"), a new approach for combining the query facilities of traditional databases with existing search engines on the Web. WSQ, for *Web-Supported (Database) Queries*, leverages results from Web searches to enhance SQL queries over a relational database. DSQ, for *Database-Supported (Web) Queries*, uses information stored in the database to enhance and explain Web searches. This paper focuses primarily on WSQ, describing a simple, low-overhead way to support WSQ in a relational DBMS, and demonstrating the utility of WSQ with a number of interesting queries and results. The queries supported by WSQ are enabled by two *virtual tables*, whose tuples represent Web search results generated dynamically during query execution. WSQ query execution may involve many high-latency calls to one or more search engines, during which the query processor is idle. We present a lightweight technique called *asynchronous iteration* that can be integrated easily into a standard sequential query processor to enable concurrency between query processing and multiple Web search requests. Asynchronous iteration has broader applications than WSQ alone, and it opens up many interesting query optimization issues. We have developed a prototype implementation of WSQ by extending a DBMS with virtual tables and asynchronous iteration; performance results are reported.

1 Introduction

Information today is decidedly split between structured data stored in traditional databases and the huge amount of unstructured information available over the World-Wide Web. Traditional relational, object-oriented, and object-relational databases operate over well-structured, typed data, and languages such as SQL and OQL enable expressive ad-hoc queries. On the Web, millions of hand-written and automatically-generated HTML pages form a vast but unstructured amalgamation of information. Much of the Web data is indexed by search engines, but search engines support only fairly simple keyword-based queries.

In this paper we propose a new approach that combines the existing strengths of traditional databases and Web searches into a single query system. *WSQ/DSQ* (pronounced "wisk-disk") stands for *Web-Supported (Database) Queries/Database-Supported (Web) Queries*. WSQ/DSQ is not a new query language. Rather, it is a practical way to exploit existing search engines to augment SQL queries over a relational database (WSQ), and for using a database to enhance and explain Web searches (DSQ). The basic architecture is shown in Figure 1. Each WSQ/DSQ instance queries one or more traditional databases via SQL, and keyword-based Web searches are routed to existing search engines. Users interacting with WSQ/DSQ can pose queries that seamlessly combine Web searches with traditional database queries.

*This work was supported by the National Science Foundation under grant IIS-9811947 and by NASA Ames under grant NCC2-5278.

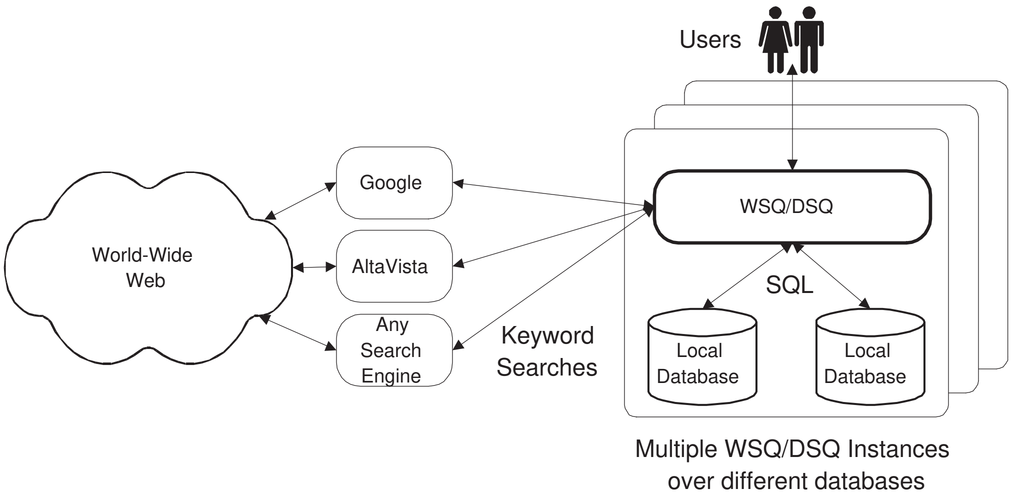

Figure 1: Basic WSQ/DSQ architecture

As an example of WSQ (Web-Supported Database Queries), suppose our local database has information about all of the U.S. states, including each state's population and capital. WSQ can enhance SQL queries over this database using Web search engines to pose the following interesting WSQ queries (fully specified in Section 3.1):

- Rank all states by how often they are mentioned by name on the Web.
- Rank states by how often they appear, normalized by state population.
- Rank states by how often they appear on the Web near the phrase "four corners".
- Which state capitals appear on the Web more often than the state itself?
- Get the top two URLs for each state.
- If Google and AltaVista both agree that a URL is among the top 5 URLs for a state, return the state and the URL.

WSQ does not perform any "magic" interpretation, cleaning, or filtering of data on the Web. WSQ enables users to write intuitive SQL queries that automatically execute Web searches relevant to the query and combine the search results with the structured data in the database. With WSQ, we can easily write interesting queries that would otherwise require a significant amount of programming or manual searching.

DSQ (Database-Supported Web Queries) takes the converse approach, enhancing Web keyword searches with information in the database. For example, suppose our database contains information about movies, in addition to information about U.S. states. When a DSQ user searches for the keyword phrase "scuba diving", DSQ uses the Web to correlate that phrase with terms in the known database. For example, DSQ could identify the states and the movies that appear on the Web most often near the phrase "scuba diving", and might even find state/movie/scuba-diving triples (e.g., an underwater thriller filmed in Florida). DSQ can be supported using the system and techniques we present in this paper, but we focus primarily on Web-supported queries (WSQ), leaving detailed exploration of DSQ for future work.

WSQ is based on introducing two *virtual tables*, WebPages and WebCount, to any relational database. A virtual table is a program that "looks" like a table to a query processor, but returns dynamically-generated tuples rather than tuples stored in the database. We will formalize our virtual tables in Section 3, but for now it suffices to think of WebPages as an infinite table that contains, for each possible Web search expression, all of the URLs returned by a search engine for that expression. WebCount can be thought of as an aggregate view over WebPages: for each possible Web search expression, it contains the total number of URLs returned by a search engine for that expression. We use WebPages_AV and WebCount_AV to denote the virtual tables corresponding to the AltaVista search engine, and we can have similar virtual tables for Google or any other search engine. By referencing these virtual tables in a SQL query, and assuring that the virtual columns defining the search expression are always bound during processing, we can answer the example queries above, and many more, with SQL alone.

While the details of WSQ query execution will be given later, it should be clear that many calls to a search engine may be required by one query, and it is not obvious how to execute such queries efficiently given typical search engine latency. One possibility is to modify search engines to accept specialized calls from WSQ database systems, but in this paper we instead show how small modifications to a conventional database query processor can exploit properties of existing search engines.

When query processing involves many search engine requests, the key observations are:

- The latency for a single request is very high.
- Unless it explicitly supports parallelism, the query processor is idle during the request.
- Search engines (and the Web in general) can handle many concurrent requests.

Thus, for maximum efficiency, a query processor must be able to issue many Web requests concurrently while processing a single query. As we will discuss in Section 4, traditional (non-parallel) query processors are not designed to handle this requirement. We might be able to configure or modify a parallel query processor to help us achieve this concurrency. However, parallel query processors tend to be high-overhead systems designed for multiprocessor computers, geared towards large data sets and/or complex queries. In contrast, the basic problem of issuing many concurrent Web requests within a query has a more limited scope that does not require traditional parallelism for a satisfactory solution. To support our WSQ framework, we introduce a query execution technique called *asynchronous iteration* that provides low-overhead concurrency for external virtual table accesses and can be integrated easily into conventional relational database systems.

The main contributions of this paper are:

- A formalization of the WebPages and WebCount virtual tables and their integration into SQL, with several examples illustrating the powerful WSQ queries enabled by this approach, and a discussion of support for such virtual tables in existing systems (Section 3).

- *Asynchronous iteration*, a technique that enables non-parallel relational query processors to execute multiple concurrent Web searches within a single query (Section 4). Although we discuss asynchronous iteration in the context of WSQ, it is a general query processing technique applicable to other scenarios as well, and it opens up interesting new query optimization issues.

- Experimental results from our WSQ prototype (Section 5), showing that asynchronous iteration can speed up WSQ queries by a factor of 10 or more.

2 Related Work

Several approaches have been proposed for bridging the divide between structured databases and the unstructured Web. *Wrappers* are used in many systems to make information in Web pages act logically as database elements, e.g., [AK97, CDSS98, HGMC+97, PGGMU95, RS97]. Wrappers are a useful means of enabling expressive queries over data that was not necessarily designed for querying, and wrappers also facilitate the integration of data from multiple, possibly heterogeneous sources, e.g., [CGMH+94, LRO96, RAH+96]. Unfortunately, wrappers over Web data tend to be labor-intensive and brittle, often requiring "screen-scraping" to parse HTML into meaningful structures. *Semistructured* data models [Abi97, Bun97, PGMW95], in particular *XML* [XML97], provide some hope for introducing structure into Web data and queries [DFF+99, GMW99]. However, we believe that vast amounts of information will remain in HTML, and will continue to be queried through search engines such as AltaVista, Google, and others. New query languages have been proposed for dynamically navigating and extracting data from the Web, e.g., [KS95, MMM97]. Our work differs in that we do not invent a new query language, and our queries combine results from Web searches with traditional structured data.

The techniques we know of that most closely relate to WSQ/DSQ are reported in [CDY95] and [DM97]. Written before the explosion of the World-Wide Web, [CDY95] focuses on execution and optimization techniques for SQL queries integrated with keyword-based external text sources. There are three main differences between [CDY95] and our work. First, they aim to minimize the number of external calls, rather than providing a mechanism to launch the calls concurrently. Nevertheless, some of techniques they propose are complementary to our framework and could be incorporated. Second, they assume that external text sources return search results as unordered sets, which enables optimizations that are not always possible when integrating SQL with (ranked) Web search results. Third, some of their optimizations are geared towards external text searches that return small (or empty) results, which we believe will be less common in WSQ given the breadth of the World-Wide Web. [DM97] discusses approaches for coupling a search engine with SQL, again without focusing on the World-Wide Web. A query rewrite scheme is proposed for automatically translating queries that call a search engine via a user-defined predicate into more efficient queries that integrate a search engine as a virtual table. While we also use a virtual table abstraction for search engines, [DM97] does not address the issue of high-latency external sources, which forms the core of much of this paper.

The integration of external relations into a cost-based optimizer for LDL is discussed in [CGK89]. The related, more general problem of creating and optimizing query plans over external sources with limited access patterns and varying query processing capabilities has been considered in work on data integration, e.g., [HKWY97, LRO96, Mor88, RSU95, YLGMU99]. In contrast, we focus on a specific scenario of one type of external source (a Web search engine) with known query capabilities. [BT98] addresses the situation where an external source may be unavailable at a particular time: a query over multiple external sources is rewritten into a sequence of incremental queries over subsets of sources, such that the query results can be combined over time to form the final result. Although the asynchronous iteration technique we introduce shares the general spirit of computing portions of a query and filling in remaining values later, our technique operates at a much finer (tuple-level) granularity, it does not involve query rewriting, and the goal is to enable concurrent processing of external requests rather than handling unavailable sources.

As will be seen in Section 4, we rely on *dependent joins* to supply bindings to our virtual tables when we integrate Web searches into a SQL query. Hence, previous work on optimizing and efficiently executing queries involving dependent joins is highly applicable. A general-purpose query optimization algorithm in the presence of dependent joins is provided in [FLMS99]. A caching technique that can be applied to improve the implementation of dependent joins is discussed in [HN96].

Much of the research discussed in this section is either preliminary or complementary to WSQ/DSQ. To the best of our knowledge, no previous work has taken our approach of enabling a non-parallel database engine to support many concurrent calls to external sources during the execution of a single query.

3 Virtual Tables in WSQ

For the purpose of integrating Web searches with SQL, we can can abstract a Web search engine through a virtual WebPages table:

WebPages(SearchExp, T1, T2, ..., Tn, URL, Rank, Date)

where SearchExp is a parameterized string representing a Web search expression. SearchExp uses "%1", "%2", and so on to refer to the values that are bound during query processing to attributes T1, T2, ..., Tn, in the Unix printf or scanf style. For example, if SearchExp is "%1 near %2", T1 is bound to "Colorado" and T2 is bound to "Denver", then the corresponding Web search is "Colorado near Denver". For a given SearchExp and given bindings for T1, T2, ... Tn, WebPages contains 0 or more (virtual) tuples, where attributes URL, Rank, and Date are the values returned by the search engine for the search expression. The first URL returned by the search engine has Rank = 1, the second has Rank = 2, and so on. It is only practical to use WebPages in a query where SearchExp, T1, T2, ..., Tn are all bound, either by default (discussed below), through equality with a constant in the Where clause, or through an equi-join. In other words, these attributes can be thought of as "inputs" to the search engine. Furthermore, because retrieving all URLs for a given search expression could be extremely expensive (requiring many additional network requests beyond the initial search), it is prudent to restrict Rank to be less than some constant (e.g., Rank $<$ 20), and this constant also can be thought of as an input to the search engine.

A simple but very useful view over WebPages is:

WebCount(SearchExp, T1, T2, ..., Tn, Count)

where Count is the total number of pages returned for the search expression. Many Web search engines can return a total number of pages immediately, without delivering the actual URLs. As we will see, WebCount is all we need for many interesting queries.

Note that for both tables, not only are tuples generated dynamically during query processing, but the number of columns is also a function of the given query. That is, a query might bind only column T1 for a simple keyword search, or it might bind T1, T2, ..., T5 for a more complicated search. Thus, we really have an infinite family of infinitely large virtual tables. For convenience in queries, SearchExp in both tables has a default value of "%1 near %2 near %3 near ... near %n".[1] For WebPages, if no restriction on Rank is included in the query, currently we assume a default selection predicate Rank $<$ 20 to prevent "runaway" queries.

Note also that virtual table WebCount could be viewed instead as a scalar function, with input parameters SearchExp, T1, T2, ..., Tn, and output value Count. However, since WebPages and other virtual tables can be more general than scalar functions—they can "return" any number of columns and any number of rows—our focus in this paper is on supporting the general case.

3.1 Examples

In this section we use WebPages and WebCount to write SQL queries for the examples presented informally in Section 1. In addition to the two virtual tables, our database contains one regular stored table:

States(Name, Population, Capital)

For each query, we restate it in English, write it in SQL, and show a small fraction of the actual result. The population values used for Query 2 are 1998 estimates from the U.S. Census Bureau [Uni98]. Queries 1–5 were issued to AltaVista (altavista.com), and Query 6 integrates results from both AltaVista and Google (google.com). All searches were performed in October 1999.[2]

Query 1: Rank all states by how often they appear by name on the Web.

```
Select Name, Count
From States, WebCount
Where Name = T1
Order By Count Desc
```

Note that we are relying on the default value of "%1" for WebCount.SearchExp. The first five results are:

<California, 4995016> <Washington, 4167056> <New York, 3764513>
<Texas, 2724285> <Michigan, 1621754> ...

Readers might be unaware that Texas and Michigan are the 2nd and 8th most populous U.S. states, respectively. Washington ranks highly because it is both a state and the U.S. capital; a revised query could exploit search engine features to avoid some false hits of this nature, but remember that our current goal is not one of "cleansing" or otherwise improving accuracy of Web searches.

Query 2: Rank states by how often they appear, normalized by state population.

```
Select Name, Count/Population As C
From States, WebCount
Where Name = T1
Order By C Desc
```

[1] For search engines such as Google that do not explicitly support the "near" operator, we use "%1 %2 ... %n" as the default.

[2] It turns out that repeated identical Web searches may return slightly different results, so your results could exhibit minor differences.

Now, the first five results are:

<Alaska, 1149> <Washington, 733> <Delaware, 690> <Hawaii, 635> <Wyoming, 603> ...

Query 3: Rank states by how often they appear on the Web near the phrase "four corners".

```
Select Name, Count
From States, WebCount
Where Name = T1 and T2 = 'four corners'
Order By Count Desc
```

Recall that "%1 near %2" is the default value for WebCount.SearchExp when T1 and T2 are bound. There is only one location in the United States where a person can be in four states at once: the "four corners" refers to the point bordering Colorado, New Mexico, Arizona, and Utah. Note the dramatic dropoff in Count between the first four results and the fifth:

<Colorado, 1745> <New Mexico, 1249> <Arizona, 1095> <Utah, 994> <California, 215> ...

Query 4: Which state capitals appear on the Web more often than the state itself?

```
Select Capital, C.Count, Name, S.Count
From States, WebCount C, WebCount S
Where Capital = C.T1 and Name = S.T1 and C.Count > S.Count
```

In the following (complete) results, we again see some limitations of text searches on the Web—more than half of the results are due to capitals that are very common in other contexts, such as "Columbia" and "Lincoln":

<Atlanta, 1053868, Georgia, 958280> <Lincoln, 669059, Nebraska, 385991>
<Boston, 1409828, Massachusetts, 1006946> <Jackson, 1120655, Mississippi, 662145>
<Pierre, 663310, South Dakota, 283821> <Columbia, 1668270, South Carolina, 540618>

Query 5: Get the top two URLs for each state. We omit query results since they are not particularly compelling.

```
Select Name, URL, Rank
From States, WebPages
Where Name = T1 and Rank <= 2
Order By Name, Rank
```

Query 6: If Google and AltaVista both agree that a URL is among the top 5 URLs for a state, return the state and the URL.

```
Select Name, AV.URL
From States, WebPages_AV AV, WebPages_Google G
Where Name = AV.T1 and Name = G.T1 and AV.Rank <= 5 and G.Rank <= 5 and AV.URL = G.URL
```

Surprisingly, Google and AltaVista only agreed on the relevance of 4 URLs:

<Indiana, www.indiana.edu/copyright.html> <Louisiana, www.usl.edu>
<Minnesota, www.lib.umn.edu> <Wyoming, www.state.wy.us/state/welcome.html>

3.2 Support for virtual tables in existing systems

Both IBM DB2 and Informix currently support virtual tables in some form. We give a quick overview of the support options in each of these products, summarizing how we can modify our abstract virtual table definitions to work on such systems. (We understand that Oracle also expects to support virtual tables in a future release.) See [RP98] for more information about support for virtual tables in database products.

In DB2, virtual tables are supported through *table functions*, which can be written in Java or C [IBM]. A table function must export the number and names of its columns. Hence, DB2 cannot support a variable number of columns, so we would need to introduce a family of table functions WebPages1, WebPages2, etc. to handle the different possible number of arguments, up to some predetermined maximum; similarly for WebCount. To the query processor, a table function is an iterator supporting methods *Open*, *GetNext*, and *Close*. Currently, DB2 provides no "hooks" into the query processor for pushing selection predicates into a table function. At first glance, this omission apparently prevents us from implementing WebPages or WebCount, since both tables logically contain an infinite number of tuples and require selection conditions to become finite. However, DB2 table functions support parameters that can be correlated to the columns of other tables in a From clause. For example, consider:

```
Select R.c1, S.c3
From R, Table(S(R.c2))
```

In this query, S is a table function that takes a single parameter. DB2 will create a new table function iterator for each tuple in R, passing the value of c2 in that tuple to the *Open* method of S. (DB2 requires that references to S come after R in the From clause.) With this feature, we can implement WebPages and WebCount by requiring that SearchExp and the n search terms are supplied as table function parameters, either as constants or using the From clause join syntax shown in the example query above. In the case of WebPages, we must pass the restriction on Rank as a parameter to the table function as well.

Informix supports virtual tables through its *virtual table interface* [SBH98]. Unlike DB2, Informix provides hooks for a large number of functions that the DBMS uses to create, query, and modify tables. For example, in Informix a virtual table scan can access the associated Where conditions, and therefore can process selection conditions. However, the Informix query processor gives no guarantees about join ordering, even when virtual tables are involved, so we cannot be sure that the columns used to generate the search expression are bound by the time the query processor tries to scan WebPages or WebCount. Thus, Informix currently cannot be used to implement WebPages or WebCount (although, as mentioned earlier, WebCount could be implemented as a user-defined scalar function, which is supported in Informix).

4 WSQ Query Processing

Even with an ideal virtual table interface, traditional execution of queries involving WebCount or WebPages would be extremely slow due to many high-latency calls to one or more Web search engines. As mentioned in Section 2, [CDY95] proposes optimizations that can reduce the number of external calls, and caching techniques [HN96] are important for avoiding repeated external calls. But these approaches can only go so far—even after extensive optimization, a query involving WebCount or WebPages must issue some number of search engine calls.

In many situations, the high latency of the search engine will dominate the entire execution time of the WSQ query. Any traditional non-parallel query plan involving WebCount or WebPages will be forced to issue Web searches sequentially, each of which could take one or more seconds, and the query processor is idle during each request. Since Web search engines are built to support many concurrent requests, a traditional query processor is making poor use of available resources.

Thus, we want to find a way to issue as many concurrent Web searches as possible during query processing. While a parallel query processor (such as Oracle, Informix, Gamma [DGS+90], or Volcano [Gra90]) is a logical option to evaluate, it is also a heavyweight approach for our problem. For example, suppose a query requires 50 independent Web searches (for 50 U.S. states, say). To perform all 50 searches concurrently, a parallel query processor must not only dynamically partition the problem in the correct way, it must then launch 50 query threads or processes. Supporting concurrent Web searches during query processing is a problem of restricted scope that does not require a full parallel DBMS.

In the remainder of this section we describe *asynchronous iteration*, a new query processing technique that can be integrated easily into a traditional non-parallel query processor to achieve a high number of concurrent Web searches with low overhead. As we will discuss briefly in Section 4.2, asynchronous iteration is in fact a general query processing technique that can be used to handle a high number of concurrent calls to any external sources. (In future work, we plan to compare asynchronous iteration against the performance of a parallel query processor over a range of queries involving many calls to external sources.) As described in the following subsections, asynchronous iteration also opens up interesting new query optimization problems.

4.1 Asynchronous Iteration

Let us start with an example. Suppose in our relational database we have a simple table Sigs(Name), identifying the different ACM Special Interest Groups, called "Sigs"—e.g., SIGMOD, SIGOPS, etc. Now we want to use WebCount to rank the Sigs by how often they appear on the Web near the keyword "Knuth":[3]

```
Select *
From Sigs, WebCount
Where Name = T1 and T2 = 'Knuth'
Order By Count Desc
```

[3] Incidentally, the results (in order) from AltaVista are: SIGACT, SIGPLAN, SIGGRAPH, SIGMOD, SIGCOMM, SIGSAM. For all other Sigs, Count is 0.

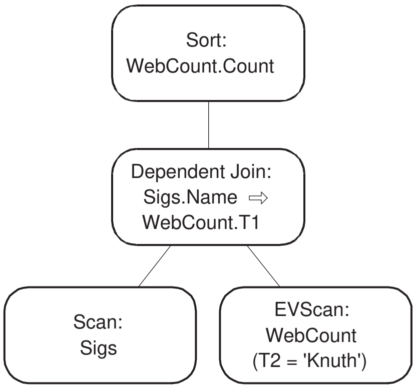

Figure 2: Query plan for Sigs $\bowtie$ WebCount

Figure 2 shows a possible query plan for this query. For this plan, and for all other plans in this paper, we assume an iterator-based execution model [Gra93] where each operator in the plan tree supports *Open*, *GetNext*, and *Close* operations. The *Dependent Join* operator requires each *GetNext* call to its right child to include a binding from its left child, thus limiting the physical join techniques that can be used to those of the nested-loop variety (although work in [HN96] describes hashing and caching techniques that can improve performance of a dependent join). The *EVScan* operator is an external virtual table scan. We assume that we are working with a query processor that can produce plans of this sort—with dependent joins and scans of virtual tables—such as IBM DB2 (recall Section 3.2).

Without parallelism, EVScan performs a sequence of Web searches during execution of this query plan (one for each *GetNext* call), and the query processor may be idle for a second or more each time. Intuitively, we would like the query processor to issue many Web searches simultaneously, without the overhead of a parallel query processor. For this small data set—37 tuples for the 37 ACM Sigs—we would like to issue all 37 requests at once. To achieve this behavior we propose asynchronous iteration, a technique involving three components:

1. A modified, asynchronous version of EVScan that we call *AEVScan*.
2. A new physical query operator called *ReqSync* (for "Request Synchronizer"), which waits for asynchronously launched calls to complete.
3. A global software module called *ReqPump* (for "Request Pump"), for managing all asynchronous external calls.

The general idea is that we modify a query plan to incorporate asynchronous iteration by replacing EVScans with AEVScans and inserting one or more ReqSync operators appropriately within the plan. AEVScan and ReqSync operators both communicate with the global ReqPump module. No other query plan operators need to be modified to support asynchronous iteration.

Now we walk through the actual behavior of asynchronous iteration using our example. Consider the query plan in Figure 3. In comparison to Figure 2, the EVScan has been replaced by an AEVScan, the ReqSync operator has been added, and the global ReqPump is used. When tuples are constructed during query processing, we allow any attribute value to be marked with a special *placeholder* that serves two roles:

1. The placeholder indicates that the attribute value (and thus the tuple it's a part of) is incomplete.

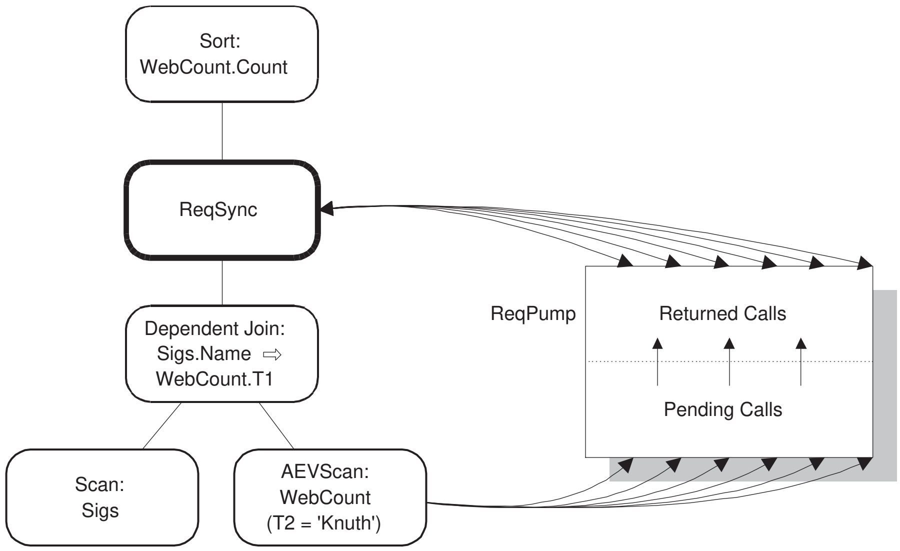

Figure 3: Asynchronous iteration

2. The placeholder identifies a pending ReqPump call associated with the missing value—that is, the pending call that will supply the true attribute value when the call finishes.

Recall that all of our operators, including AEVScan and ReqSync, obey a standard iterator interface, including *Open*, *GetNext*, and *Close* methods. We now discuss in turn how the operators in our example query plan work.

The Scan and Sort operators are oblivious to asynchronous iteration. The Dependent Join (hereafter DJ) is a standard nested-loop operator that also knows nothing about asynchronous iteration. Now consider the AEVScan. When DJ gets a new tuple from Sigs, it calls *Open* on AEVScan and then calls *GetNext* with Sigs.Name. AEVScan in turn contacts ReqPump and registers an external call C with T1 = Sigs.Name and T2 = 'Knuth'. (C is a unique identifier for the call.) ReqPump is a module that issues asynchronous network requests and stores the responses to each request as they return. In the case of call C, the returned data is simply a value for Count; ReqPump stores this value in a hash table *ReqPumpHash*, keyed on C. To achieve concurrency, as soon as AEVScan registers its call with ReqPump, it returns to DJ (as the result of *GetNext*) one WebCount tuple T where the Count attribute contains as a placeholder the call identifier C. DJ combines T with Sigs.Name and returns the new tuple to its parent (ReqSync).

Now let us consider the behavior of ReqSync. When its *Open* method is called from above by Sort, ReqSync calls *Open* on DJ below and then calls *GetNext* on DJ until exhaustion, buffering all returned (incomplete) tuples inside ReqSync. We choose this full-buffering implementation for the sake of simplicity, and we will revisit this decision momentarily. ReqSync needs to coordinate with ReqPump to fill in placeholders before returning tuples to its parent. The problem is a variation of the standard "producer/consumer" synchronization problem. Each ReqPump call is a producer: when a call C' completes (and its data is stored in ReqPumpHash), ReqPump signals to the consumer (ReqSync) that the data for C' is available. When signaled by ReqPump, ReqSync locates the incomplete tuple containing C' as a placeholder (using its own local hash table), and replaces C' with the Count value retrieved from ReqPumpHash. When ReqSync's *GetNext* method is called from above, if ReqSync has no completed tuples then it must wait for the next signal from ReqPump before it can return a tuple to its parent. Note that in the general case, tuples that do not depend on pending ReqPump calls may pass directly through a ReqSync operator.

In our simple implementation of ReqSync's *Open* method, all (incomplete) tuples generated by DJ are buffered inside ReqSync before ReqSync can return any (completed) tuples to its parent. In the case of very large joins it might make sense for ReqSync to make completed tuples available to its parent before exhausting execution of its child subplan. As with query execution in general, the question of materializing temporary results versus returning tuples as they become available is an optimization issue [GMUW00].

As we will show in Section 5, asynchronous iteration can improve WSQ query performance by a factor of 10 or more over a standard sequential query plan. However, there are still three important lingering issues that we will discuss in Sections 4.3, 4.4, and 4.5, respectively:

1. As seen in our example, an external call for WebCount always generates exactly one result tuple. But a call for WebPages may produce any number of tuples, including none, and the number of generated tuples is not known until the call is complete.

2. When a query plan involves more than one AEVScan, we must account for the possibility that an incomplete tuple buffered in ReqSync could contain placeholders for two or more different pending ReqPump calls.

3. We need to properly place ReqSync operators in relation to other query plan operators, both to guarantee correctness and maximize concurrency.

Monitoring and controlling resource usage is also an important issue when we use asynchronous iteration. So far we have assumed that during query execution we can safely issue an unbounded number of concurrent search requests. Realistically, we need to regulate the amount of concurrency to prevent a search engine from being inundated with an "unwelcome" number of simultaneous requests. Similarly, we may want to limit the total number of concurrent outgoing requests to prevent WSQ from exhausting its own local resources, such as network bandwidth. It is quite simple to modify ReqPump to handle such limits: we need only add one counter to monitor the total number of active requests, and one counter for each external destination. An administrator can configure each counter as desired. When a call is registered with ReqPump but cannot be executed because of resource limits, the call is placed on a queue. As resources free up, queued calls are executed.

4.2 Applicability of asynchronous iteration

Before delving into details of the three remaining technical issues outlined in the previous subsection, let us briefly consider the broader applicability of asynchronous iteration. Although this paper describes asynchronous iteration in the specific context of WSQ, the technique is actually quite general and applies to most situations where queries depend on values provided by high-latency, external sources. More specifically, if an external source can handle many concurrent requests, or if a query issues independent calls to many different external sources, then asynchronous iteration is appropriate. Our WSQ examples primarily illustrate the first case (many concurrent requests to one or two search engines). As an example of the second case,

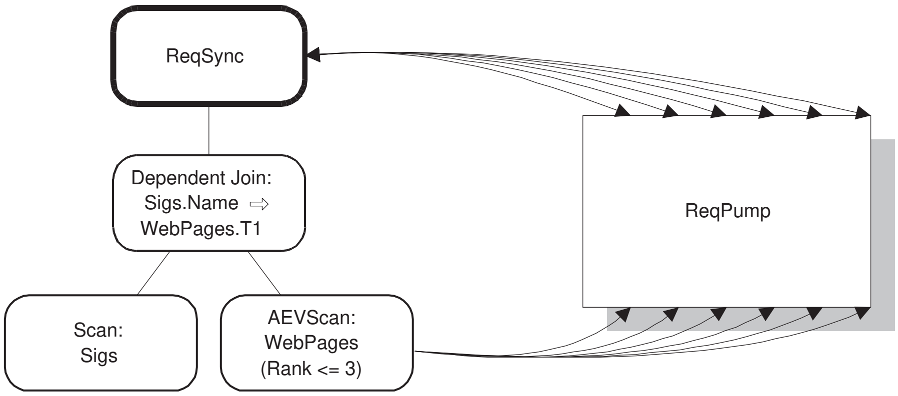

Figure 4: Query plan for Sigs ⋈ WebPages

asynchronous iteration could be used to implement a Web crawler: given a table of thousands of URLs, a query over that table could be used to fetch the HTML for each URL (for indexing and to find the next round of URLs). In this scenario, WSQ can exploit all available resources without burdening any external sources.

As mentioned earlier, if we try to use a parallel query processor to achieve the high level of concurrency offered by asynchronous iteration, then we would need to partition tables dynamically into many small fragments and spawn many query threads or processes. Issuing many threads can be expensive. For example, the highest performance Web servers do not use one thread per HTTP request; rather, many network requests are handled asynchronously by an event-driven loop within a single process [PDZ99]. By implementing the ReqPump module of asynchronous iteration in a similar manner, we can enable many simultaneous calls with low overhead. Nonetheless, as future work we plan to conduct experiments comparing the performance of asynchronous iteration against a parallel DBMS for managing concurrent calls to external sources.

4.3 ReqSync tuple generation or cancellation

The previous example (Figure 3) was centered on a dependent join with WebCount, which always yields exactly one matching tuple. But WebPages, and any other virtual table in general, may return any number of tuples for given bindings—including none. Because we want AEVScan to return from a *GetNext* call without waiting for the actual results, we always begin by assuming that exactly one tuple joins, then "patch" our results in ReqSync.

Consider the following query, which retrieves the top 3 URLs for each Sig.

```
Select *
From Sigs, WebPages
Where Name = T1 and Rank <= 3
```

For each Sig, joining with WebPages may generate 0, 1, 2, or 3 tuples. Assume a simple query plan as shown in Figure 4. As in our previous example, AEVScan will use ReqPump to generate 37 search engine calls, and ReqSync will initially buffer 37 tuples. Now consider what happens for a tuple T, waiting in a ReqSync buffer for a call C to complete. When C returns, there are three possibilities:

1. If C returns no rows, then ReqSync deletes T from its buffer.

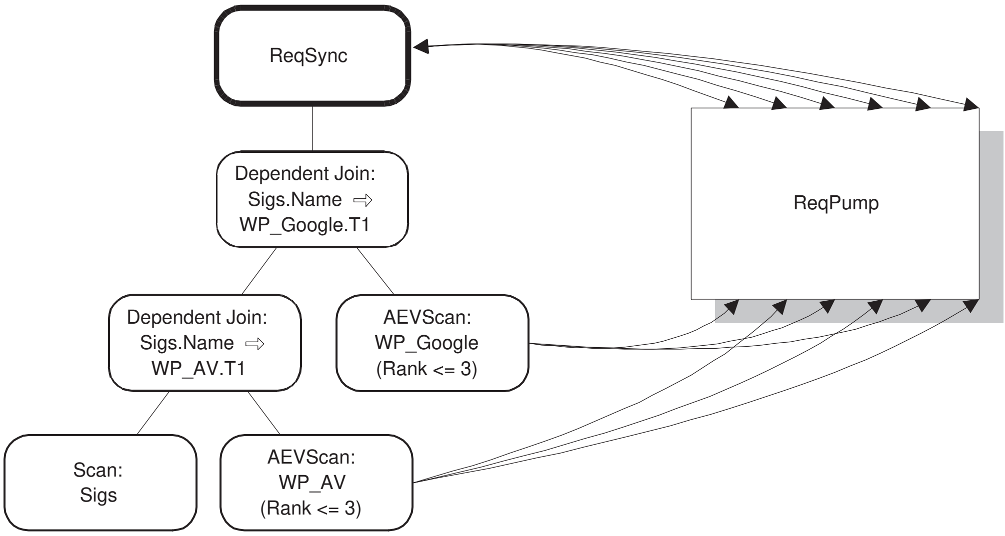

Figure 5: Query plan for Sigs ⋈ WebPages_AV ⋈ WebPages_Google

2. If C returns 1 row, then ReqSync fills in the attribute values for T as generated by C.
3. If C returns n rows, where $n > 1$, then ReqSync dynamically creates $n - 1$ additional copies of T, and fills in the attribute values accordingly.

In our example, since all Sigs are mentioned on at least 3 Web pages, 111 tuples are ultimately produced by ReqSync.

4.4 Handling multiple AEVScans

Now let us consider query plans involving multiple AEVScans. For example, the following query finds the top 3 URLs for each Sig from two different search engines.[4]

```
Select *
From Sigs, WebPages_AV AV, WebPages_Google G,
Where Name = AV.T1 and Name = G.T1 and AV.Rank <= 3 and G.Rank <= 3
```

Figure 5 shows a query plan that maximizes concurrent requests. Note that there is only one ReqSync operator, not one for each AEVScan. The placement and merging of ReqSync operators is discussed in Section 4.5. In this plan, the bottom Dependent Join will generate 37 tuples, each with placeholders identifying a ReqPump call for WebPages_AV. The upper join will augment each of these tuples with additional placeholders corresponding to a ReqPump call for WebPages_Google. Hence, ReqSync will buffer 37 incomplete tuples, each one with placeholders for two different ReqPump calls.

The algorithm for tuple cancellation, completion, and generation at the end of Section 4.3 applies in this case as well, with a slight nuance: dynamically copied tuples (case 3 in the algorithm) may proliferate references to pending calls. For example, suppose one of the incomplete tuples T in the ReqSync buffer is

[4]The query actually finds all combinations of the top 3 URLs from each search engine, but it nonetheless serves to illustrate the point of this section.

waiting for the completion of two calls, indicated by two different placeholders: one for call C_A to AltaVista and the other for call C_G to Google. If C_A returns first, with 3 tuples, then ReqSync will make two additional copies of T. When copying T, references to pending call C_G are also copied. Once C_G returns, all tuples referencing C_G must be updated.

4.5 Query plan generation

Recall that converting a query plan to use asynchronous iteration has two parts: (1) EVScan operators are converted to AEVScans, and (2) ReqSync operators are added to the plan. In this section we describe an algorithm for placing ReqSync operators within plans. Our primary goal is to introduce a correct and relatively simple algorithm that: (1) attempts to maximize the number of concurrent Web searches; (2) attempts to maximize the amount of query processing work that can be performed while waiting for Web requests to be processed; and (3) is easy to integrate into existing query compilers. ReqSync operators can significantly alter the cost of a query plan, and the effects on query execution time will often depend on the specific database instance being queried, as well as the results returned by search engines. Fully addressing cost-based query optimization in the presence of asynchronous iteration is an important, interesting, and broad problem that is beyond the scope of this paper. We intend to focus on optimization in future work.

We assume that the optimizer can generate plans with dependent joins [FLMS99] and EVScans, but knows nothing about asynchronous iteration; a plan produced by the optimizer is the input to our algorithm. We continue to assume an iterator model for all plan operators. We now describe the three steps in our placement of ReqSync operators: *Insertion*, *Percolation*, and *Consolidation*.

4.5.1 ReqSync Insertion

Recall that we first convert each EVScan operator in our input plan P to an asynchronous AEVScan. Next, a ReqSync operator is inserted directly above each AEVScan. More formally, for each AEVScan$_i$ in P, we insert ReqSync$_i$ into P as the parent of AEVScan$_i$. The previous parent of AEVScan$_i$ becomes the parent of ReqSync$_i$. This transformation is obviously correct since no operations occur between each asynchronous call and the blocking operator that waits for its completion.

4.5.2 ReqSync Percolation

Next, we try to move ReqSync operators up the query plan. Intuitively, each time we pull up a ReqSync operator we are increasing the amount of query processing work that can be done before blocking to wait for external calls to complete. Sometimes we can rewrite the query plan slightly to enable ReqSync pull-up. For example, if the parent of a ReqSync is a selection predicate that depends on attribute values filled in by ReqSync, we can pull ReqSync higher by pulling the selection predicate up first. Similarly, if a join depends on values filled in by ReqSync, we can rewrite the join as a selection over a cross-product and move the ReqSync above the cross-product.

Our actual algorithm is based on the notion of an operator O *clashing* with a ReqSync operator, in which case we cannot pull ReqSync above O. Let ReqSync$_i.A$ denote the set of attributes whose values are filled in by the ReqSync_i operator as ReqPump calls complete, i.e., the attributes whose values are substituted with placeholders by AEVScan_i. We say that O *clashes* with ReqSync_i iff:

1. O depends on the value of any attribute in $\text{ReqSync}_i.A$, or
2. O removes any attribute in $\text{ReqSync}_i.A$ via projection, or
3. O is an aggregation or existential operator

Case 1 is clear: an operator clashes if it needs the attributes filled in by ReqSync_i to continue processing. Case 2 is a bit more subtle. If we project away placeholders before the corresponding calls are complete, then tuple cancellation or generation (Section 4.3) cannot take place properly, and extra tuples or incorrect numbers of duplicates may be returned. Case 3 is similar to case 2: aggregation (e.g., Count) and existential quantification require an accurate tally of incoming tuples.

For each ReqSync_i in the plan, we repeatedly pull ReqSync_i above any non-clashing operators. If an operator O does clash, we check to see if O is a projection or selection; if so, we can pull O above its parent first. Otherwise, if O is a clashing join, we rewrite it as a selection over a cross-product. Other similar rewrites are possible. For example, a set union operator must examine each complete tuple to perform duplicate elimination; we can rewrite this clashing operator as a "Select Distinct" over a non-clashing bag union operator. Our percolation algorithm clearly terminates since operators are only pulled up the plan. Also, the order in which we percolate ReqSync operators does not matter—the only potential effect is a different final ordering between adjacent ReqSync operators, something that is made irrelevant by ReqSync Consolidation, which we discuss next. We will illustrate the percolation algorithm through examples momentarily.

4.5.3 ReqSync Consolidation

After percolation, we may find that two or more ReqSync operators are now adjacent in the plan. At this point we can merge adjacent ReqSync operators since they perform the same overall function, and a single ReqSync operator can manage multiple placeholder values in tuples as discussed in Section 4.4. When merging ReqSync_i with ReqSync_j, $\text{ReqSync}_i.A \cup \text{ReqSync}_j.A$ is the set of attributes that must be filled in by the new ReqSync operator.

4.5.4 Plan generation examples

We now show three examples demonstrating our ReqSync placement algorithm. We point out the performance gains asynchronous iteration can provide, along with some potential pitfalls of our current algorithm.

Example 1: Figure 6 shows how our ReqSync placement algorithm generates the query plan we saw earlier in Figure 5 for the Sigs $\bowtie$ WebPages_AV $\bowtie$ WebPages_Google query. We omit ReqPump from these (and all remaining) query plans. Figure 6(a) shows the input to the algorithm, a simple left-deep query plan without asynchronous iteration. Figure 6(b) shows the plan after ReqSync Insertion: the EVScans are converted to AEVScans and a ReqSync operator is inserted directly above each EVScan. Figure 6(c) shows the plan after ReqSync Percolation. We first move ReqSync_1 above both dependent joins, since neither join depends on any values returned by WebPages_AV (i.e., URL, Date, Rank). ReqSync_2 is then pulled above its parent

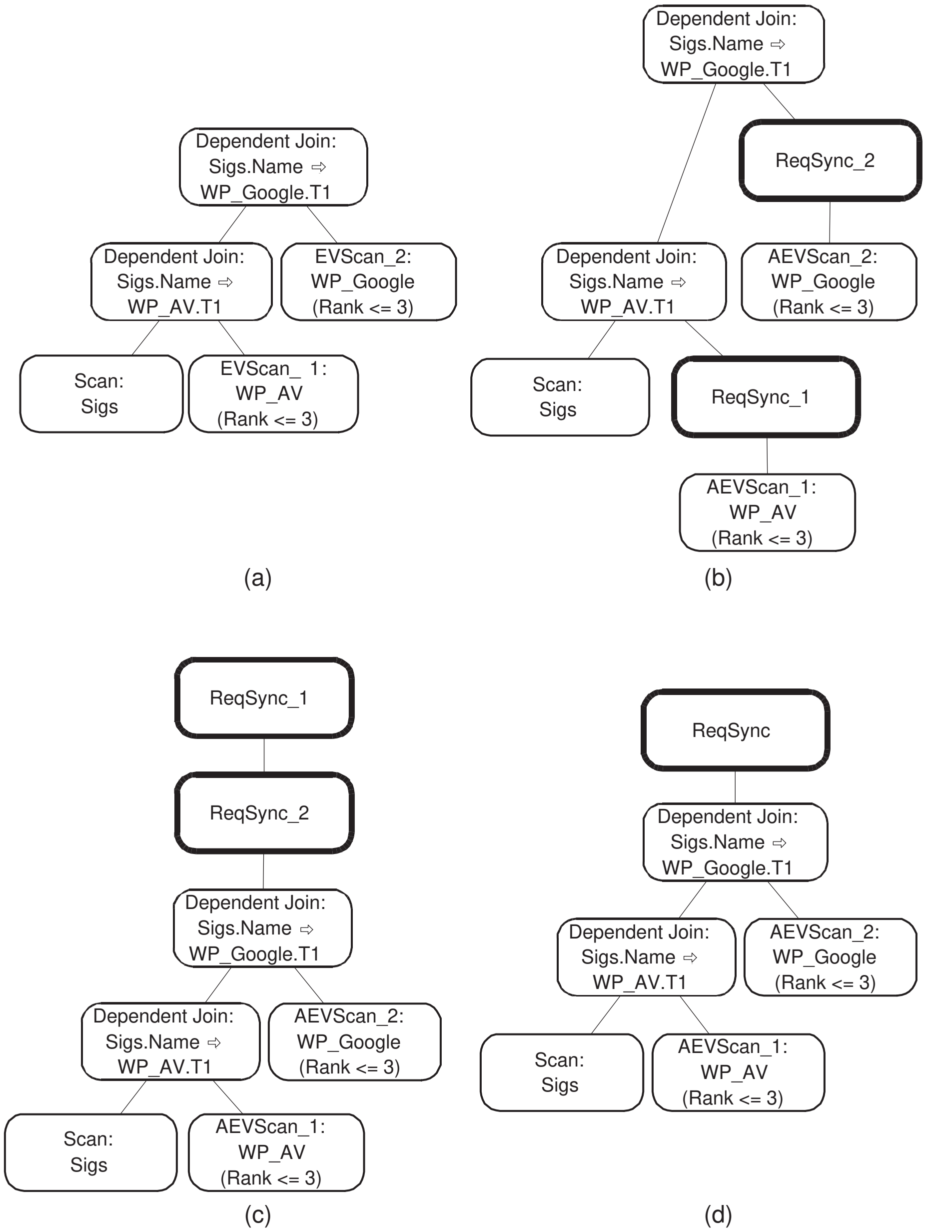

Figure 6: Generating the query plan for Sigs ⋈ WebPages_AV ⋈ WebPages_Google in Figure 5

dependent join. The final plan after ReqSync Consolidation is shown in Figure 6(d). With this plan, the query processor can process all 74 external calls (37 Sigs per join) concurrently.

This example demonstrates some interesting advantages of asynchronous iteration over possible alternatives. First, one might consider simply modifying the dependent join operator to work in parallel: change the dependent join to launch many threads, each one for joining one left-hand input tuple with the right-hand EVScan. While this approach will provide maximal concurrency for many simple queries, it prevents concurrency among requests from multiple dependent joins: the query processor will block until the first join completes. Another approach, as discussed in Section 4.2, is to use a (modified) parallel query processor for this query. However, performing both dependent joins in parallel requires a nontrivial rewrite to transform our 2-join plan into a 3-join plan where both dependent joins are children of a final "merging" join. □

Example 2: Consider the following query, where a cross-product with a meaningless table R is introduced for illustrative purposes:

```
Select *
From Sigs, WebCount_AV AV, R, WebCount_Google G
Where Name = AV.T1 and Name = G.T1
```

Figure 7(a) shows the result of running our ReqSync placement algorithm over a left-deep input plan in which the cross-product with R is performed between the two virtual table dependent joins. With or without asynchronous iteration, this input plan is problematic: by performing the cross-product before the join with WebCount_Google, a straightforward dependent join implementation will send $|\mathsf{R}|$ identical calls to Google for each Sig. Thus, incorporating a local cache of search engine results is very important for such a plan. Furthermore, when using asynchronous iteration with the plan in Figure 7(a), the cross-product with table R will generate $|\mathsf{R}|$ copies of the incomplete tuples from WebCount_AV that must be buffered and then patched by ReqSync. Depending on the data, it may be preferable to use two ReqSync operators as shown in Figure 7(b). By doing so, we reduce the total number of attribute values to be patched by $|\mathsf{Sigs}| \cdot (|\mathsf{R}| - 1)$, or roughly a factor of 2 for reasonably large $|\mathsf{R}|$. On the down side, we will block after the first join, preventing us from concurrently issuing the Web requests for WebCount_Google. Had the cross-product with R been placed last in the original input plan, another alternative would be to place a single ReqSync operator above the dependent joins but below the cross-product.

This contrived example serves to illustrate the challenging query optimization problems that arise when we introduce AEVScan and ReqSync operators. Still, in many cases our simple ReqSync placement algorithm does perform well, as we will see in Section 5. □

Example 3: As a final example suppose that we also have a table CSFields(Name) containing computer science fields (e.g., "databases", "operating systems", "artificial intelligence", etc.). Consider the following query, which finds URLs that are among the top 5 URLs for both a Sig and a CSField.

```
Select S.URL
From Sigs, WebPages S, CSFields, WebPages C
Where Sigs.Name = S.T1 and CSFields.Name = C.T1 and S.Rank <= 5 and
        C.Rank <= 5 and S.URL = C.URL
```

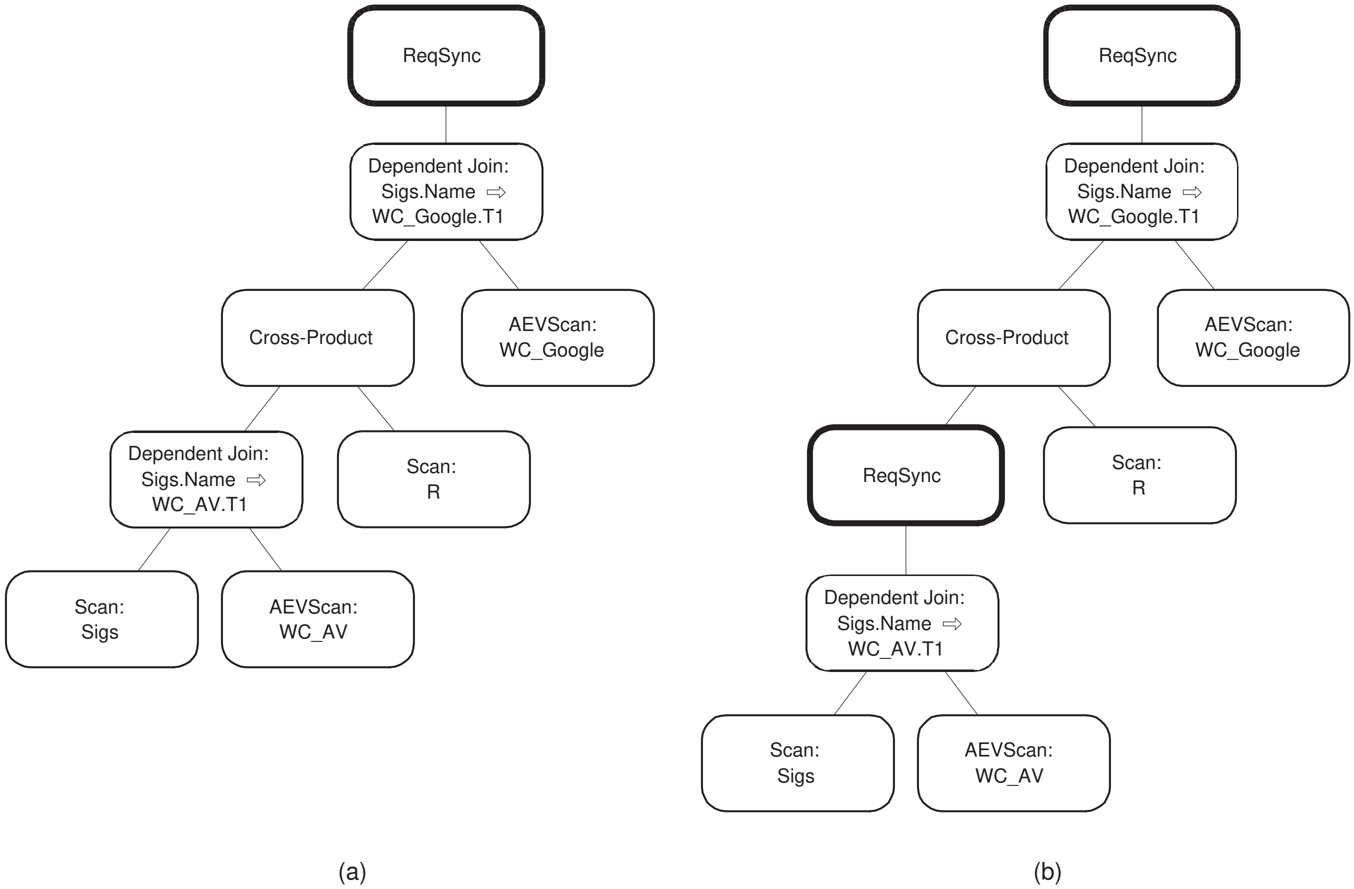

Figure 7: A query plan mixing two dependent joins with a cross-product

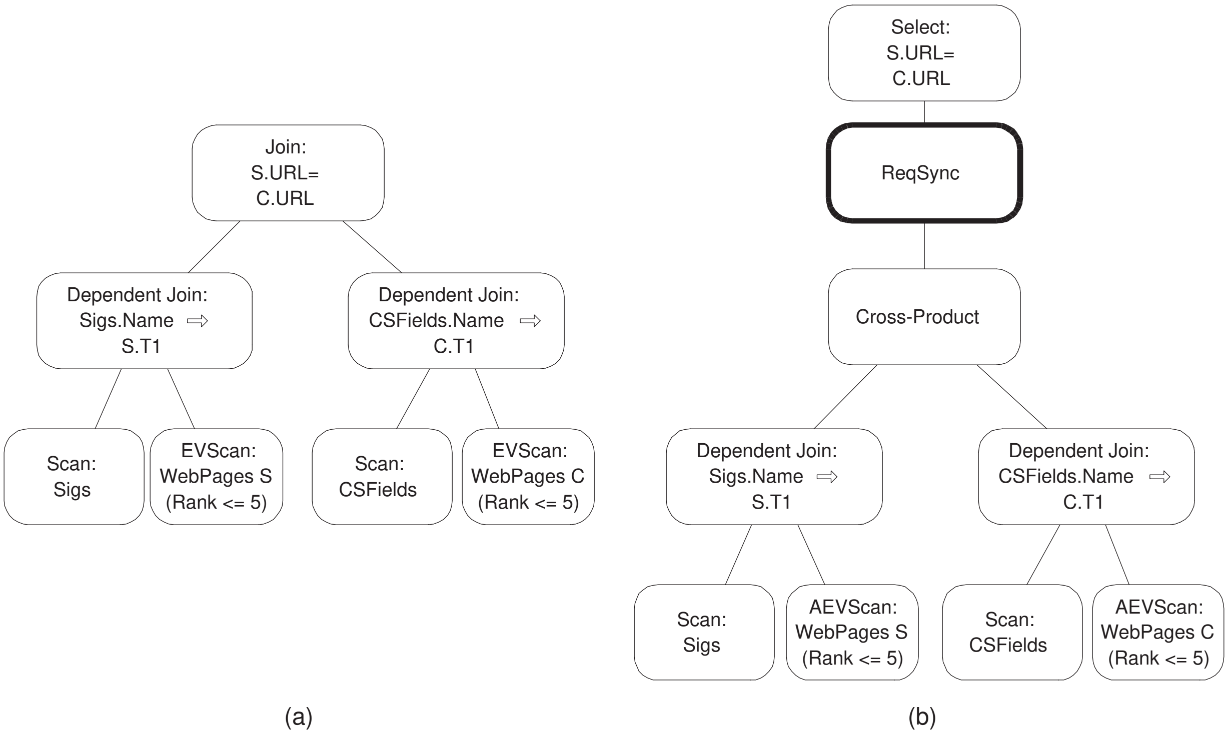

Figure 8: Generating the query plan for query over Sigs and CSFields

An input query plan is shown in Figure 8(a). Note that the input plan is bushy, and the join at the the root of the plan may well be implemented as a sort-merge or hash join. After inserting the two ReqSync operators, we first pull them above the dependent joins. To pull the ReqSyncs above the upper join, we rewrite the join into a selection over a cross-product, as described in Section 4.5.2. (Because the join depends on attributes supplied by WebPages, we can't pull the ReqSync above it without the rewrite.) Figure 8(b) shows the final plan.

In this query, given that the Sigs and CSFields tables are tiny, rewriting the join as a cross-product is a big performance win: it enables the query processor to execute all external calls (from both the left and right subplans) concurrently. However, in other situations, such as if the cross-product is huge, this specific rewrite could be a mistake.

This example illustrates one more important issue. Suppose that a Sig does not have any URLs on a given search engine. Indeed, assume for the moment that all Sigs have no URLs, so all Sig tuples generated will ultimately be canceled. In that case, pulling the ReqSync operator up as in Figure 8(b) results in an unnecessary cross-product between placeholder tuples for CSFields and WebPages, since ultimately the cross-product (and therefore the join) will be empty. In the general case, because AEVScan always returns exactly one matching tuple before the final result is known, a plan could perform unnecessary work—work that would not be done if the query processor waited for the true Web search result before continuing. □

To summarize, the above examples demonstrate how our ReqSync placement algorithm focuses on maximizing the number of concurrent external calls for any given query plan. If external calls dominate query execution time, then asynchronous iteration can provide dramatic performance improvements, as we demonstrate in Section 5. Nevertheless, there are several potential performance pitfalls that are best addressed by a complete cost-based query optimizer incorporating asynchronous iteration:

- Manipulating query plans to use asynchronous iteration may change their relative performance. Given two equivalent input plans A and B, where $Cost(A) < Cost(B)$, there is no guarantee that the asynchronous version of A will remain cheaper than the asynchronous version of B.
- The ReqSync operator buffers tuples, possibly proliferates them, and fills in missing attribute values. In some situations it is possible that the amount of work required by ReqSync offsets the advantages of asynchronous iteration.
- Asynchronous iteration assumes non-empty join results and continues processing, patching results later as necessary. If join results do turn out to be empty, then our "optimistic" approach will have performed more work than necessary.
- In order to pull ReqSync operators higher, we may move or rewrite operators in the input query plan, such as replacing joins with selections over cross-products. Additional work induced by these rewrites could offset the benefit of additional concurrency.

5 Implementation and Experiments

We have integrated the two WSQ virtual tables and our asynchronous iteration technique into a homegrown relational database management system called *Redbase*. (Redbase is constructed by students at Stanford in a course on DBMS implementation.) Redbase supports a subset of SQL for select-project-join queries, and it includes a page-level buffer and iterator-based query execution. However, it was not designed to be a high-performance system: the only available join technique is nested-loop join, and there is no query optimizer although users can specify a join ordering manually. Nevertheless, Redbase is stable and sophisticated enough to support the experiments in this section, which demonstrate the potential of asynchronous iteration. Our experiments show the considerable performance improvement of running WSQ queries with asynchronous iteration as opposed to conventional sequential iteration.

Measuring the performance of WSQ queries has some inherent difficulties. First, performance of a search engine such as AltaVista can fluctuate considerably depending on load and network delays beyond our control. Second, because of caching behavior beyond our control, repeated searches with identical keyword expressions may run far faster the second (and subsequent) times. To mitigate these issues, we waited at least two hours between queries that issue identical searches, which we verified empirically is long enough to eliminate caching behavior. Also, we performed our experiments late at night when the load on search engines is low and, more importantly, consistent.

In order to run many experiments without waiting hours between each one, we use *template* queries and instantiate multiple versions of them that are structurally similar but result in slightly different searches being issued. Consider the following template.

	Synchronous (secs)	Asynchronous (secs)	Improvement
Template 1			
Run 1 (8 queries)	23.13	3.88	6.0x
Run 2 (8 other queries)	32.8	3.5	9.4x
Template 2			
Run 1 (8 queries)	70.75	5.25	13.5x
Run 2 (8 other queries)	64.25	5.13	12.5x
Template 3			
Run 1 (8 queries)	122.5	6.25	19.6x
Run 2 (8 other queries)	76.13	4.63	16.4x

Table 1: Experimental results

Template 1:

```
Select Name, Count
From States, WebCount
Where Name = T1 and WebCount.T2 = V1
```

V1 represents a constant that is chosen from a pool of different common constants, such as "computer", "beaches", "crime", "politics", "frogs", etc. For our experiments, we created 8 instances of the template by choosing 8 different constants from the pool. After timing all queries using asynchronous iteration, we waited two hours and then timed all queries using the standard query processor. For corroboration, we repeated the test with 8 new query instances.

The results for this template (and the two below) are shown in Table 1. For each template, we list the results of two runs. The times listed are the average execution time in seconds for the 8 queries, with and without asynchronous iteration. AltaVista is used for the first two templates; the third uses both AltaVista and Google. Experiments were conducted on a Sun Sparc Ultra-2 (2 x 200Mhz) 256MB RAM machine running SunOS 5.6. The computer is connected to the Internet via Stanford University's network.

Template 2:

```
Select Name, Count, URL, Rank
From States, WebCount, WebPages
Where Name = WebCount.T1 and WebCount.T2 = V1 and
    Name = WebPages.T1 and WebPages.T2 = V2 and WebPages.Rank <= 2
```

In this query template, we issue two searches for each tuple in States, one for WebCount and one for WebPages. When instantiating the template we wanted to ensure that V1 $\neq$ V2, so we selected 16 distinct constants to create 8 query instances. In our prototype system, the join order is always specified by the order of tables in the From clause, so for this query we joined States with WebCount, then joined the result with WebPages. Results are shown in Table 1.

Template 3: The following template is similar to the example in Section 4.4 (Figure 5), with the added constant V1. Again, we created 8 queries by instantiating V1 with constants, and results are shown in Table 1.

```
Select Name, AV.URL, G.URL
From Sigs, WebPages_AV AV, WebPages_Google G,
Where Name = AV.T1 and Name = G.T1 and AV.Rank <= 3 and G.Rank <= 3 and
   AV.T2 = V1 and G.T2 = V1
```

Our results show clearly that asynchronous iteration can improve the performance of WSQ queries by a factor of 10 or more. Of course, all of the example queries here are over very small local tables, so network costs dominate. These results in effect illustrate the best-case improvement offered by asynchronous iteration. For queries involving more complex local query processing over much larger relations, the speedup may be less dramatic, and the results of any such experiment would be highly dependent on the sophistication of the database query processor (independent of asynchronous iteration). Further, as illustrated in Section 4, complex queries may introduce optimization decisions that could have a significant impact on performance. In future work we plan a comprehensive study of query optimization incorporating asynchronous iteration, including additional experiments and performance comparisons to alternate approaches such as parallel query processing.

We have created a simple interface that allows users to pose limited queries over our WSQ implementation. Please visit http://www-db.stanford.edu/wsq.

Acknowledgments

We thank Serge Abiteboul for his contributions to early WSQ/DSQ discussions, and Jason McHugh for helpful comments on an initial draft of this paper. We also thank Berthold Reinwald and Paul Brown for information about IBM DB2 and Informix, respectively.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[AK97] N. Ashish and C.A. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[BT98] P. Bonnet and A. Tomasic. Partial answers for unavailable data sources. In *Proceedings of the Third International Conference on Flexible Query Answering Systems (FQAS)*, pages 43–54, Roskilde, Denmark, May 1998.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, Tucson, Arizona, May 1997. Tutorial.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 177–188, Seattle, Washington, June 1998.

[CDY95] S. Chaudhuri, U. Dayal, and T. Yan. Join queries with external text sources: Execution and optimization techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 410–422, San Jose, California, 1995.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 195–203, Amsterdam, The Netherlands, August 1989.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis project: Integration of heterogeneous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, October 1994.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW8)*, Toronto, Canada, 1999.

[DGS+90] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[DM97] S. Dessloch and N. Mattos. Integrating SQL databases with content-specific search engines. In *Proceedings of the Twenty-Third Internatial Conference on Very Large Databases*, pages 276–285, Athens, Greece, August 1997.

[FLMS99] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311–322, Philadelphia, Pennsylvania, June 1999.

[GMUW00] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25–30, Philadelphia, Pennsylvania, June 1999.

[Gra90] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, New Jersey, May 1990.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 10–17, Tucson, Arizona, May 1997.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the Twenty-Third Internatial Conference on Very Large Databases*, pages 276–285, Athens, Greece, August 1997.

[HN96] J.M. Hellerstein and J.F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 423–434, Montreal, Canada, June 1996.

[IBM] IBM DB2 Universal Database SQL Reference Version 6. ftp://ftp.software.ibm.com/ps/products/db2/info/vr6/pdf/letter/db2s0e60.pdf.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pages 54–65, Zurich, Switzerland, September 1995.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-Second International Conference on Very Large Databases*, pages 251–262, Bombay, India, September 1996.

[MMM97] A.O. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54–67, April 1997.

[Mor88] K.A. Morris. An algorithm for ordering subgoals in NAIL! In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 82–88, Austin, Texas, 1988.

[PDZ99] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 161–186, Singapore, December 1995.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[RAH$^+$96] M.T. Roth, M. Arya, L.M. Haas, M.J. Carey, W.F. Cody, R. Fagin, P.M. Schwarz, J. Thomas II, and E.L. Wimmers. The Garlic project. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 557, Montreal, Canada, June 1996.

[RP98] B. Reinwald and H. Pirahesh. SQL open heterogenous data access. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 506–507, Seattle, Washington, June 1998.

[RS97] M.T. Roth and P.M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of the Twenty-Third Internatial Conference on Very Large Databases*, pages 266–275, Athens, Greece, August 1997.

[RSU95] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 105–112, San Jose, California, May 1995.

[SBH98] M. Stonebraker, P. Brown, and M. Herbach. Interoperability, distributed applications and distributed databases: The virtual table interface. *Data Engineering Bulletin*, 21(3):25–33, 1998.

[Uni98] United States Bureau of the Census. State population estimates and demographic components of population change: July 1, 1997 to July 1, 1998. http://www.census.gov/ population/estimates/state/st-98-1.txt, December 1998.

[XML97] World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/ TR/WD-xml-lang-970331.html, December 1997.

[YLGMU99] R. Yerneni, C. Li, H. Garcia-Molina, and J. Ullman. Optimizing large join queries in mediation systems. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 348–364, Jerusalem, Israel, January 1999.