Data Mining Applications in ATLAS

Carlo Zaniolo*

*Course Notes for CS240B

Classifiers

The play tennis example, and its vertical version.

RID	Outlook	Temp	Humidity	Wind	Play
1	Sunny	Hot	High	Weak	No
2	Rain	Mild	High	Weak	Yes
3	Overcast	Hot	High	Weak	Yes
4	Sunny	Hot	High	Strong	No
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	Yes
7	Overcast	Cool	Normal	Strong	No
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

RID	col	val	YorN
1	1	Rain	No
1	2	Mild	No
1	3	High	No
1	4	Strong	No
2	1	Rain	Yes
2	2	Mild	Yes
2	3	High	Yes
2	4	Weak	Yes
3	1	. . .	
. . .		. . .	

Table Functions

Dissemble a relation into column/value pairs:

```
FUNCTION dissemble (v1 Int, v2 Int, v3 Int, v4 Int, YorN Int)
                : (col Int, val Int, YorN Int);
{     INSERT INTO RETURN
      VALUES (1, v1, YorN), (2, v2, YorN),
             (3, v3, YorN), (4, v4, YorN);
}
```

Then you write the classifier and call it as follows:

```
SELECT classify(0, p.RID, d.Col, d.Val, d.YorN)
FROM PlayTennis AS p,
 TABLE(dissemble(Outlook,Temp, Humidity, Wind, Play)) AS d;
```

Naive Bayesian Classifiers

- For each pair (column, value) count the positives and negatives and store them in a `summary` table:

col	**val**	**Yc**	**Nc**
1	Rain	14	11
1	Sunny	21	13
1	Overcast	23	17
2	Hot	14	28
2	...	...	...

- Also tally up the positives and negatives
- These operations are done in one pass
- The resulting table is all is needed to classify a new tuple. Example: (Sunny, Hot, ...).

Decision Tree Classifiers

We start by computing the SUMMARY table
Then select a column for splitting using the gini index:

- We start by computing the SUMMARY table.
- Compute the Gini index g for each column.
  1. Gini for each value in the column:
  $f_p = p/(p+n),\ f_n = n/(p+n),\ g = 1 - f_p^2 - f_n^2$
  2. Gini for the column: $Gini = f_1 \times g_1 + f_2 \times g_2 + \ldots$

col	val	Yc	Nc
1	Rain	14	11
1	Sunny	21	13
1	Overcast	23	17
2	Hot	14	28
2	...	...	...

For instance for column 1 we have three values: rain, sunny, overcast (in the implementation these values are coded as integers).

Decision Tree Classifiers–cont.

- Find the column with the least $Gini$ (using **minpair**) and store it in **mincol**.
- Reclassify the tuples by splitting each class according to their values in column c. An unique id must be generated for the new class.
- Recursive invocation (but homogenous classes and columns with only one value are not split)

A Scalable Decision Tree Classifier

```
AGGREGATE classify(iNode Int, RecId Int, iCol Int,
                   iValue Int, iYorN Int)
{ TABLE treenodes(RecId Int, Node Int, Col Int,
                  Value Int, YorN Int);
  TABLE mincol(Col Int);
  TABLE summary(Col Int, Value Int, Yc Int, Nc Int,
                INDEX {Col,Value});
  TABLE ginitable(Col Int, Gini Int);
  INITIALIZE : ITERATE : {
    INSERT INTO treenodes
      VALUES(RecId, iNode, iCol, iValue, iYorN);
    UPDATE summary
      SET Yc=Yc+iYorN, Nc=Nc+1-iYorN
      WHERE Col = iCol AND Value = iValue;
    INSERT INTO summary
      SELECT iCol, iValue, iYorN, 1-iYorN
      WHERE SQLCODE¡¿0;
  }
```

A Scalable Decision Tree Classifier—Cont

```
TERMINATE : {
  INSERT INTO ginitable
    SELECT Col, sum((Yc*Nc)/(Yc+Nc))/sum(Yc+Nc)
    FROM summary
    GROUP BY Col;
    HAVING count(Value)>1
        AND sum(Yc)>0 AND sum(Nc)>0;
  INSERT INTO mincol
    SELECT minpair(Col, Gini) FROM ginitable;
  INSERT INTO result
    SELECT iNode, Col FROM mincol;
  /* Call classify() recusively to partition each of its subnodes unless it is pure.*/
  SELECT classify(t.Node*MAXVALUE+m.Value+1,
             t.RecId, t.Col, t.Value, t.YorN)
    FROM   treenodes AS t,
      ( SELECT tt.RecId, tt.Value
        FROM treenodes AS tt, mincol AS m
        WHERE tt.Col=m.Col
      ) AS m
    WHERE t.RecId = m.RecId
    GROUP BY m.Value; }
}
```

Association Rules: Apriori

- Many attempts to implement frequent-item-set computations in SQL DBMS and O-R DBs have failed to produce good performance
- In-depth investigation by Sarawagi, Thomas, and Agrawal [ACM/SIGMOD 98] established this as the acid test for any SQL extension claiming to do support data mining
- Our previous system, AXL, had also failed because of poor performance
- ATLaS solved the problem via (i) table functions, (ii) in-memory tables, and (iii) better optimization techniques.

Example: Apriori Algorithm—minimun support $= 2$

$\mathcal{D}$:

TID	Items
1000	1, 3, 4
200	2, 3, 5
300	1, 2, 3, 5
400	2, 5

Scan $\mathcal{D}$

Itemset	Support
{1}	2
{2}	3
{3}	3
{4}	1
{5}	3

Filter

Itemset	Support
{1}	2
{2}	3
{3}	3
{5}	3

Build Larger Sets

$\rightarrow$

Itemset
{1,2 }
{1,3 }
{1, 5 }
{2,3 }
{2,5}
{3, 5 }

Scan $\mathcal{D}$

Itemset	Support
{1,2 }	1
{1,3 }	2
{1, 5 }	1
{2,3 }	2
{2,5 }	3
{3, 5 }	2

Filter

Itemset	Support
{1,3 }	2
{2,3 }	2
{2,5 }	3
{3, 5 }	2

Build Larger Sets

$\rightarrow$

Itemset
{2,3, 5 }

Scan $\mathcal{D}$

Itemset	Support
{2,3, 5 }	2

Filter:

Itemset	Support
{2,3, 5 }	2

Main Operations

- Scanning the database and counting occurrences
- Pruning the itemsets below the minimum support level:
- Combining frequent sets of size n into candidate larger sets of size $n+1$ [or even larger].
  Monotonicity Condition: The support level of a set is always $\leq$ *than that of every subset*
- Checking the presence of large-cardinality item sets is expensive. A prefix tree is used to solve this problem.
- In memory tables with references were used here.

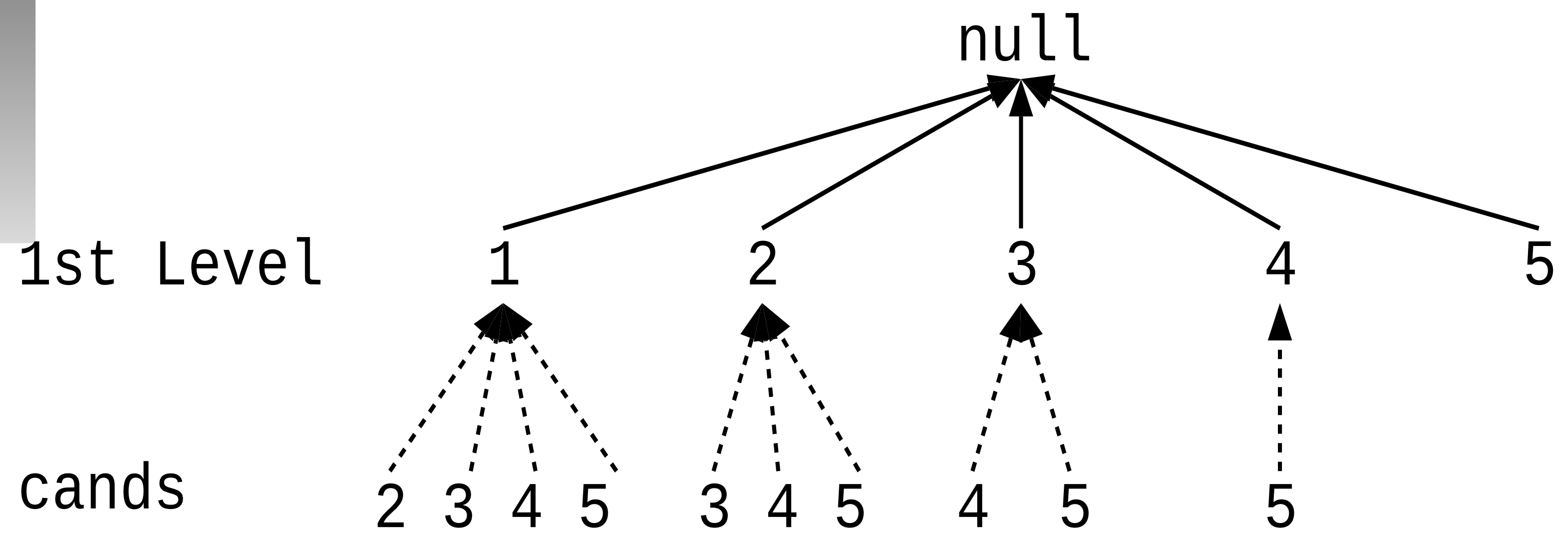
A Priori in ATLaS

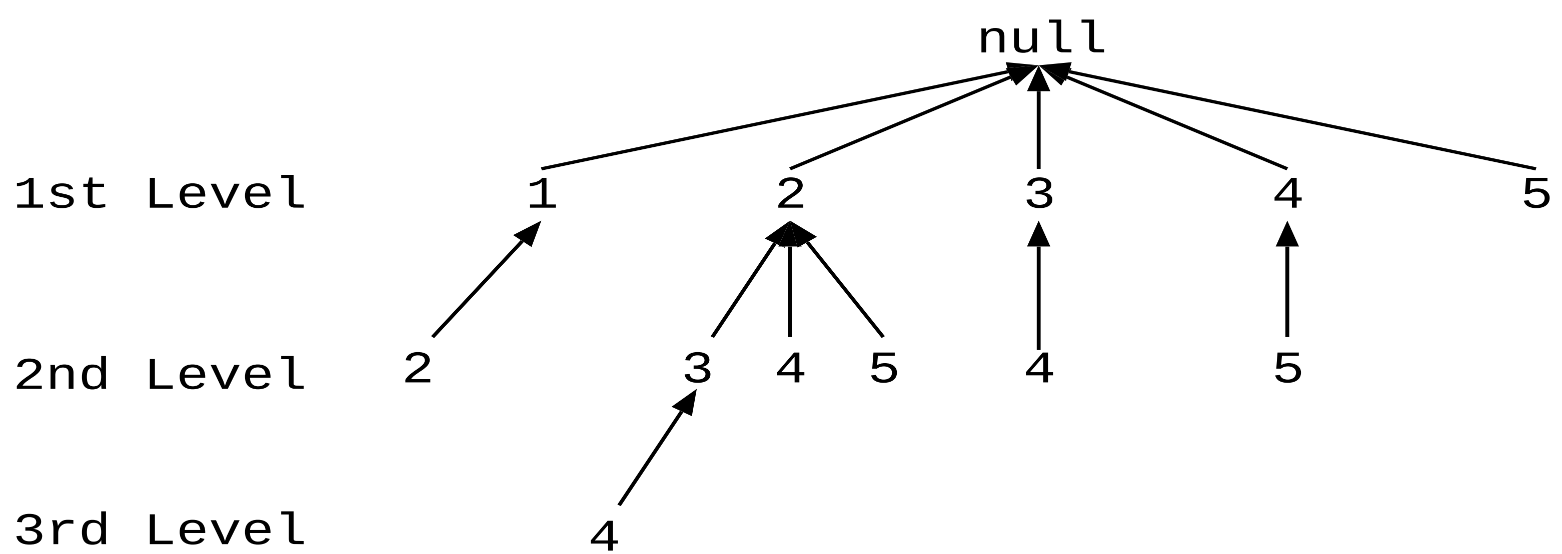
- **baskets(item** INT): A stream of transactions from the DB table

$$\mathbf{0}, 2, 3, 4, \mathbf{0}, 1, 2, 3, 4, \mathbf{0}, 3, 4, 5, \mathbf{0}, 1, 2, 5, \mathbf{0}, 2, 4, 5$$

- The frequent itemsets in the prefix tree: **trie**
- The tuples in **cands** hold an item, **cit**, a reference, **trieref**, to a leaf node of the trie, and a the count **freqcount**, for the set.

The Prefix Tree





Main ATLaS Program for Apriori

1: TABLE **baskets(item** Int**);**

2: TABLE **trie(item** Int**, father** REF**(trie),** INDEX**(father))** MEMORY**;**

3: TABLE **cands(item** Int**, trieref** REF**(trie), freqcount** Int**,**
INDEX**(cit,trieref))** MEMORY**;**

4: TABLE **fitems(item** Int**,** INDEX**(item));**

*/*generate frequent singleton sets*/*

5: INSERT INTO **fitems**
SELECT **item** FROM **baskets** WHERE **item** $>$ **0**
GROUP BY **item** HAVING **count(*)** $\geq MinSup$**;**

/ intialize the trie to contain frequent singletons*/*

6: INSERT INTO **trie** SELECT **item, null** FROM **fitems;**

*/*self-join frequent 1-itemsets to get candidate 2-item sets*/*

7: INSERT INTO **cands**
SELECT **t1.itno, t2.OID, 0** FROM **trie** AS **t1, trie** AS **t2**
WHERE **t1.itno** $>$ **t2.itno;**

*/*Generate (k+1)-itemsets from k-itemsets. Start with k=2*/*

8: SELECT **countset(item, 2,** $MinSup$**, cands)** FROM **baskets;**

Managing the Prefix Tree

- A perfect candidate for an TRIE ADT coded in C++
- But we did it in ATLaS SQL
- Using an in-memory table using SQL3 reference type to organize the TRIE data structure
- How are reference types on in-memory tables implemented in ATLaS ...?

Performance

Name	T	I	D	size of dataset
T5.I2.D100K	5	2	100K	2.8M text stream
T10.I2.D100K	10	2	100K	5.2M text stream
T10.I4.D100K	10	4	100K	5.2M text stream
T20.I2.D100K	20	2	100K	10.1M text stream

Table 1: Benchmark Data Sets

Performance Curves

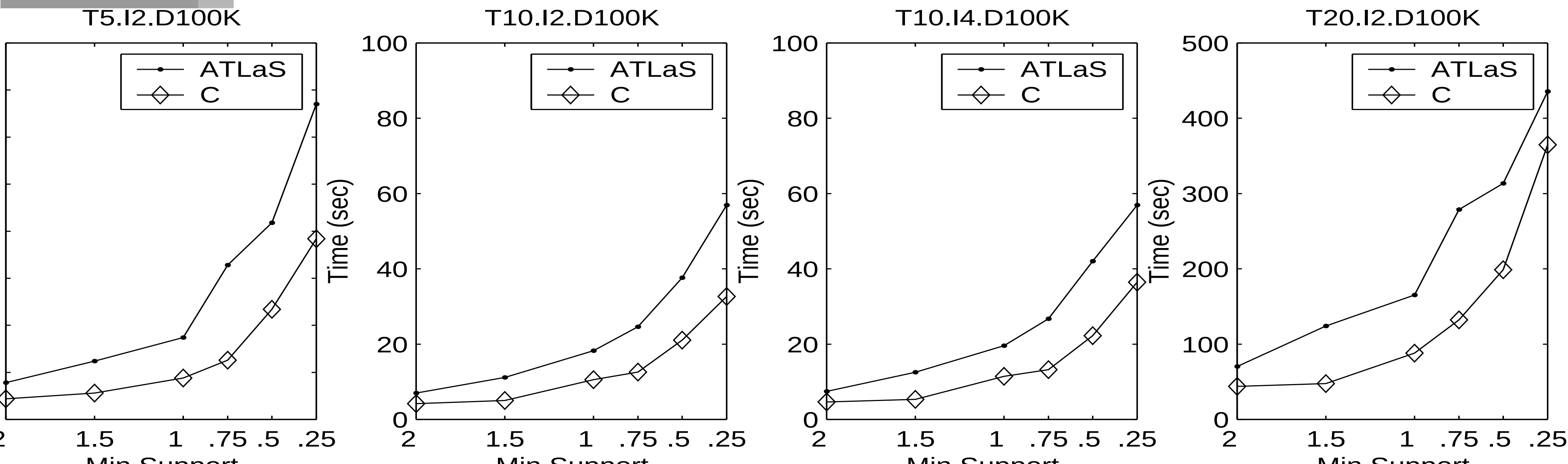


Figure 1: ATLaS vs. C implementation of Apriori

Programming in ATLaS

- Table-based programming is powerful and natural for data intensive applications
- SQL can be awkward and many extensions are possible
- But even SQL *AS IS* is adequate even for complex datamining queries and algorithms.

The ATLaS System

- ATLaS programs into C programs are compiled into C programs that Execute on the Berkeley DB record manager
- The 100 Apriori program compiles into 2,800 lines of C The system
- Other data structures (R-trees, in-memory tables) have been added using the same API.
- The system is now 54,000 lines of C++ code.

ATLaS: Conclusions

- A simple native extensibility mechanism for SQL
- More efficient than Java or PL/SQL. Effective with Data Mining Applications
- Also OLAP applications, and recursive queries, and temporal database applications
- Complements current extensibility mechanisms based on UDFs and Data Blades
- Supports and favors streaming aggregates (in SQL the default is blocking)
- Good basis for determining program properties: e.g. (non)monotonic and blocking behavior
- These are lessons that future query languages cannot easily ignore.

References

1. J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A Data Mining Query Language for Relational Databases. In Proc. 1996 SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'96), pp. 27-33, Montreal, Canada, June 1996.
2. R. Meo, G. Psaila, S. Ceri. A New SQL-like Operator for Mining Association Rules. In Proc. VLDB96, 1996 Int. Conf. Very Large Data Bases, Bombay, India, pp. 122-133.
3. T. Imielinski and A. Virmani. MSQL: a query language for database mining. Data Mining and Knowledge Discovery, 3:373-408, 1999. S.
4. S. Tsur, J. Ulman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov. Query flocks: a generalization of association rule mining. In Proc. 1998 ACM-SIGMOD, p. 1-12, 1998.
5. F. Bonchi , F. Giannotti, G. Mainetto, D. Pedreschi. A Classification-based Methodology for Planning Audit Strategies in Fraud Detection. In Proc. KDD-99, ACM-SIGKDD Int. Conf. on Knowledge Discovery & Data Mining, San Diego (CA), August 1999.
6. F. Giannotti, G. Manco, D. Pedreschi and F. Turini. Experiences with a logic-based knowledge discovery support environment. In Proc. 1999 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD 1999).
7. T. Johnson, Laks V. S. Lakshmanan, Raymond T. Ng: The 3W Model and Algebra for Unified Data Mining. VLDB 2000: 21-32.

References–Cont.

8 Haixun Wang, Carlo Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. VLDB 2000.

9 Haixun Wang, Carlo Zaniolo. ATLaS: a Native Extension of SQL for Data Mining. 2003 SIAM International Conference on Data Mining (SDM03), May 1–3, San Francisco, CA.