2
Fundamentos

Este capítulo provê o conhecimento fundamental sobre os dois pilares sobre os quais esta tese foi desenvolvida: transação e Web semântica. As seções seguintes apresentam os fundamentos necessários para o entendimento consistente dos capítulos posteriores.

2.1.
Gerenciamento de Transações

Controle de concorrência é atividade de coordenar as ações de processos que operam em paralelo, acessam dados compartilhados, e, portanto, potencialmente interferem uns nos outros. Recuperação é a atividade de garantir que falhas (de software ou hardware) não corrompam dados persistidos [Bernstein et al., 1987].

Informalmente, uma transação é uma execução de um programa que acessa um banco de dados compartilhado. O objetivo das atividades de controle de concorrência e recuperação é garantir que transações executem atomicamente [Bernstein et al., 1987], significando que:

1. cada transação acesse dados compartilhados sem interferir umas nas outras, e
2. se uma transação terminar normalmente, então todos os seus efeitos se tornam permanentes; caso contrário ela não tem nenhum efeito.

[Gray, 1981] foi um dos primeiros a descrever que as propriedades ***A****tomicidade*, ***C****onsistência*, ***I****solamento* e ***D****urabilidade* (ACID) precisam ser satisfeitas para se ter o correto processamento de transações simultâneas. Estas propriedades garantem:

- **Atomicidade**
  que transações completem como um todo ou, caso contrário, nada seja realizado. Se uma transação atualizar algum dado e não conseguir terminar com sucesso, as modificações intermediárias têm que ser revertidas. É o chamado comportamento "tudo ou nada".

- **Consistência**
  que transações levem o conteúdo do banco de dados de um estado consistente para outro também consistente. Apenas estados consistentes, que contém dados válidos, de acordo com regras ou invariantes de consistência, serão escritos no banco de dados.

- **Isolamento**
  que transações simultâneas operem em um ambiente isolado. Isto significa que mudanças intermediárias realizadas por uma transação em execução não são visíveis para outras transações. Na prática, o objetivo é garantir que o efeito da execução de um conjunto de transações simultâneas seja o mesmo de executar uma transações por vez, uma após a outra. Este comportamento é chamado escalonamento serializável e usualmente é implementado por bloqueio (*locking*, em inglês*)*.

- **Durabilidade**
  que mudanças produzidas por uma transação finalizada com sucesso sejam duráveis e não possam ser revertidas.

Caso o acrônimo ACID não seja respeitado, transações executando em paralelo podem criar resultados incorretos que podem trazer consequências severas. A seção seguinte descreve os quatro problemas que comumente ocorrem, caso o controle de concorrência seja falho ou inexistente.

2.1.1. Problemas Comuns de Concorrência

A ineficiência ou falta de controle de concorrência pode ocasionar quatro problemas comuns. São eles:

- ***Lost updates***

  Ocorrem quando duas ou mais transações selecionam o mesmo objeto e então atualizam o objeto com base no estado original selecionado. Cada transação não sabe da ocorrência das outras transações. A última atualização sobrescreve as atualizações realizadas pelas outras transações, o que resulta em perda de dados.

TIME	T_1	T_2
t1	T_{1_begin}	
t2		T_{2_begin}
t3	$read(flightseat)$	
t4		$read(flightseat)$
t5	$checkIfFree(flightseat)$	
t6		$checkIfFree(flightseat)$
t7	$flightseat = "Mr.Smith")$	
t8	$write(flightseat)$	
t9		$flightseat = "Mrs.Mayr")$
t10		$write(flightseat)$
t11	T_{1_commit}	
t12		T_{2_commit}
INITIAL: $flightseat = "free"$, RESULT: $flightseat = "Mrs.Mayr"$		

Figura 1 - *Lost Update*: Mr. Smith sem assento [Stroka, 2009]

A Figura 1 demonstra este problema na reserva de um assento em um avião. As transações T_1 e T_2 tentam, ao mesmo tempo, reservar o mesmo assento no avião. A última transação T_2 sobrescreve as mudanças de T_1, no instante t10. *Mrs. Mayr* obtém o assento ao invés de *Mr. Smith*, que teria obtido o assento se um bloqueio de escrita estivesse sido estabelecido por T_1, que começou primeiro.

- ***Dirty reads (Uncommitted Dependency)***

  Ocorrem quando uma segunda transação seleciona um objeto que está sendo atualizado por outra transação. A segunda transação está lendo dados ainda não confirmados (*uncommitted*), que podem ser modificados pela transação que está atualizando o objeto.

TIME	T_1	T_2
t1	T_{1_begin}	
t2	$read(money)$	
t3	$money = money + 400$	
t4	$write(money)$	
t5	...	
t6		T_{2_begin}
t7		$read(hotel)$
t8		$read(money)$
t9	$T_{1\_rollback}$	
t10		$money = money - 600$
t11		$hotel = hotel + 600$
t12		$write(money)$
t13		$write(hotel)$
t14		T_{2_commit}
INITIAL: $money = 300$, RESULT: $money = 100$		

Figura 2 - *Dirty Read*: Erro no saldo da conta bancária [Stroka, 2009]

Na Figura 2, um viajante, inicialmente com 300 em sua conta bancária, deseja fazer uma reserva em um hotel. Por algum motivo, a transferência executada por T_1 precisa ser desfeita, pois o valor estava errado. Antes de a transferência ser desfeita (*rolled back*) no instante t9, T_2 inicia o pagamento da reserva do hotel, movendo o valor 600 da conta do viajante para a conta do hotel. O problema é que T_2 lê o valor da conta do viajante antes de T_1 desfazer suas atualizações. O valor final da conta do viajante será 100 quando deveria ser -300, ou seja, o viajante ganhou dinheiro indevidamente.

- ***Non-repeatable reads (Inconsistent Analysis)***
  Ocorrem quando uma segunda transação acessa o mesmo objeto mais de uma vez e lê dados diferentes a cada vez. Este problema é similar ao problema de *dirty reads*, no entanto, aqui os dados lidos pela segunda transação já foram confirmados (*committed*) pela transação que fez a atualização.

TIME	T_1	T_2
t1		T_{2_begin}
t2	T_{1_begin}	
t3		$sum = 0$
t4		$read(hotel)$
t5	$read(hotel)$	
t6		$sum = sum + hotel$
t7	$hotel = hotel + 50$	
t8		$flight = read(flight)$
t9	$write(hotel)$	
t10		$sum = sum + flight$
t11	$read(fee)$	
t12	$fee = fee + 5$	
t13	$write(fee)$	
t14	T_{1_commit}	
t15		$read(fee)$
t16		$sum = sum + fee$
t17		T_{2_commit}
INITIAL: $hotel = 600,\ flight = 150,\ fee = 60,$		
RESULT: $sum = 815$		

Figura 3 - *Non-repeatable read*: Valor total incorreto [Stroka, 2009]

A Figura 3 ilustra este problema, onde uma transação calcula o valor total a ser gasto em uma viagem. Neste meio tempo, uma segunda transação atualiza o custos do hotel (instante t9) e das taxas (instante t13). O resultado final será 815, quando deveria ser 810 se T_2 terminasse antes de T_1 começar ou 865 se T_1 terminasse antes de T_2 começar, ou seja, se não houvesse interferência de uma na outra.

- ***Phantoms reads***

  Ocorrem quando um objeto que pertence a uma coleção lida por uma transação é removido ou inserido por uma segunda transação. A primeira leitura da coleção mostra um objeto que deixa de existir nas leituras subsequentes da mesma coleção, como resultado de uma remoção por uma transação diferente. Similarmente, como resultado de uma inserção por uma transação diferente, as leituras subsequentes da mesma coleção, passam a mostrar um objeto que não existia na leitura original.

TIME	T_1	T_2
t1	T_{1_begin}	
t2	$count(hotels)$ where $location = Kiel$	
t3		T_{1_begin}
t4		$insert(newhotel)$ where $location = Kiel$
t5		T_{2_commit}
t6	$count(hotels)$ where $location = Kiel$	
t7	T_{1_commit}	
INITIAL: $hotels = 0$, RESULT: $hotels = 1$		

Figura 4 - *Phantom reads*: Mesma consulta, dois resultados [Stroka, 2009]

Na Figura 4, a quantidade de hotéis localizados em *Kiel* é consultada por T_1. Em seguida, no instante t4, T_2 insere um novo hotel também localizado em *Kiel*. No instante t6, T_1 executa a mesma consulta e, para sua surpresa, se a primeira consulta retornou 0 (zero), a segunda retornou 1, sem que T_1 tenha inserido nada. Diz-se que surgiu um registro fantasma (*phantom*).

2.1.2. Bloqueio

Como descrito anteriormente, uma das quatro propriedades básicas de transações é Isolamento ou Controle de Concorrência. Tecnicamente, isto significa que a execução simultânea das transações deve ter o mesmo efeito de uma execução serial, uma após a outra, sem sobreposição. Estas execuções são chamadas de escalonamento serializáveis, significando "ter o mesmo efeito de uma execução serial". O mecanismo mais popular usado para se obter escalonamentos serializáveis é bloqueio [Bernstein e Newcomer, 2009].

Basicamente há duas abordagens de bloqueio, a saber: pessimista e otimista. As duas seções seguintes explicam cada um delas.

2.1.2.1. Bloqueio Pessimista

O objetivo da *abordagem* pessimista é prevenção de conflitos. A ideia é bastante simples:

- Cada transação reserva acesso para o dado que irá usar. Esta reserva é chamada de bloqueio (*lock*, em inglês).

- Existem bloqueios de leitura e de escrita.
- Antes de ler o dado, a transação adquiri um bloqueio de leitura no dado. Antes de escrever (alterar ou inserir) o dado, a transação adquiri um bloqueio de escrita.
- Bloqueio de leitura conflita com bloqueio de escrita, e bloqueio de escrita conflita com bloqueio de escrita. Mas bloqueio de leitura não conflita (ou é o compatível) com bloqueio de leitura. Por estas razões, bloqueio de leitura e bloqueio de escrita também são conhecidos, respectivamente, por bloqueio compartilhável e bloqueio exclusivo.
- Uma transação pode obter um bloqueio somente se nenhuma outra transação tiver um bloqueio conflitante no mesmo item de dado. Portanto, pode obter um bloqueio de leitura em *x* somente se nenhuma outra transação tiver um bloqueio de escrita em *x*. E pode obter um bloqueio de escrita em *x* somente se nenhuma outra transação tiver um bloqueio de leitura ou bloqueio de escrita em *x*.

Apesar do conceito de bloqueio ser simples, seu efeito sobre desempenho e sua corretude podem ser bastante complexos, não intuitivos, e difíceis de predizer [Bernstein e Newcomer, 2009].

Protocolo de Bloqueio em Duas Fases (2PL)

Para garantir um escalonamento serializável, o protocolo de *Bloqueio em Duas Fases* (2PL) deve ser empregado. Este protocolo diz que uma transação tem que obter todos os seus bloqueios antes de liberar qualquer um deles. Em outras palavras, uma transação não pode liberar um bloqueio e subsequentemente obter outro bloqueio. Quando as transações obedecem este protocolo, elas têm duas fases (daí o nome do protocolo): fase de crescimento, durante a qual os bloqueios são adquiridos, e fase de encolhimento, onde os bloqueios são liberados. A operação que separa as duas fases é a primeira operação de *unlock* (liberação de bloqueio), que é a primeira operação da segunda fase [Bernstein e Newcomer, 2009].

Em muitos sistemas, talvez a maioria, uma variante do 2PL, chamada *Strict* 2PL, é utilizada. No *Strict* 2PL, a fase de encolhimento somente começa quando a transação termina, sendo aceita ou abortada. A grande vantagem desta variante é que uma transação sempre lê valores escritos por uma transação já aceita. Portanto, não será necessário abortar uma transação pelo

fato dela estar usando valores que se tornaram obsoletos, devido a um *rollback*. Isto evita abortos em cascata.

Deadlock

Quando duas ou mais transações estão competindo por um mesmo bloqueio em modos conflitantes, algumas delas serão bloqueadas e terão que esperar por outras liberarem seus bloqueios. Às vezes, pode ocorrer de se configurar um conjunto de transações, onde todas as transações estejam bloqueadas, esperando pela liberação de um bloqueio por outra transação do mesmo conjunto que, portanto, também está bloqueada. Todas as transações ficarão bloqueadas eternamente. "A" esperando por "B" e "B" esperando por "A". Este é um problema famoso chamado *deadlock*. Há duas formas de tratar *deadlocks*: prevenção ou detecção/recuperação. Por exemplo, alguns sistemas quando detectam um *deadlock*, escolhem uma transação do conjunto para abortar (*rollback*), eliminando o ciclo de espera.

Starvation

Outro problema conhecido se chamada *starvation*, onde transações são preteridas indefinidamente, sem conseguir obter o bloqueio. Isto pode ocorrer, por exemplo, por causa da constante chegada de transações de maior prioridade. Outro exemplo poderia ocorrer em sistemas onde transações não são bloqueadas para esperar por bloqueios, ou seja, a transação reinicia, espera um tempo e tenta novamente obter o bloqueio. Este ciclo pode se repetir indefinidamente. Unindo os dois problemas, existe ainda um tipo especial de *starvation* chamado *livelock*, que nada mais é do que o *deadlock* nestes sistemas onde as transações não são bloqueadas, mas reiniciadas.

Granularidade

A questão que trata do tamanho do item sobre o qual os bloqueios devem ser adquiridos é chamada de granularidade do bloqueio. Quanto mais fina a granularidade, mais preciso o bloqueio pode ser, e mais paralelismo pode ser obtido. Perceba que com uma granularidade fina, não será necessário bloquear uma transação que deseja acessar o final de um arquivo, enquanto existe outra transação acessando o início do mesmo arquivo. Por outro lado, quanto mais fina a granularidade, maior será o número de bloqueios, aumentando, portanto, o *overhead* de gerenciamento de bloqueios. Quanto maior a granularidade, menor

o *overhead* para aquisição e liberação de bloqueios, porém usualmente serão bloqueados mais dados do que a transação precisa, diminuindo a concorrência.

Existe um *trade-off* fundamental entre o nível de concorrência e o *overhead* de bloqueio, dependendo da granularidade utilizada. Granularidade grossa tem menor *overhead*, mas baixa concorrência. Granularidade fina tem maior concorrência, mas maior *overhead*. Exemplos de grânulo são: todo o banco de dados, arquivos, páginas, registros, etc.

Bloqueio Multigranular

Esta seção foi escrita com base no texto apresentado em [Bernstein et al., 1987]. Maiores detalhes também podem ser encontrados em [Gray et al., 1976].

Como descrito anteriormente, granularidade não tem importância com relação à corretude, não obstante, com relação a desempenho, tem um impacto bastante considerável.

Selecionar o grânulo a ser utilizado requer um balanceamento não trivial entre *overhead* e nível de concorrência. É possível fazer melhor do que escolher um grânulo ótimo, explorando o melhor de todos os grânulos, por meio do chamado bloqueio multigranular (MGL). MGL permite cada transação utilizar o grânulo mais apropriado para seu tamanho. Transações longas, que acessam vários itens, podem utilizar grânulos mais grossos. Por exemplo, se uma transação acessa vários registros de um arquivo, ela simplesmente adquiri um bloqueio no arquivo inteiro, ao invés de bloquear cada registro individual. Transações curtas podem utilizar grânulos mais finos. Desta forma, transações longas não perdem tempo estabelecendo uma enorme quantidade de bloqueios, e transações curtas não interferem artificialmente em outras transações, estabelecendo bloqueios em porções do banco de dados que elas não acessam.

O protocolo MGL requer um *Lock Manager* que previna que duas transações estabeleçam bloqueios conflitantes em dois grânulos que se sobreponham. Este é o critério de corretude do protocolo. Por exemplo, um arquivo não pode ser bloqueado para leitura por uma transação longa se um registro deste arquivo já tiver sido bloqueado para escrita por uma transação curta. A solução satisfatória utilizada para este problema é explorar o relacionamento hierárquico natural entre bloqueios de diferentes granularidades. Este relacionamento hierárquico é representado por meio de um *lock type graph*, como o mostrado na Figura 5, onde cada aresta aponta de um tipo de dados de maior granularidade para um de menor granularidade.

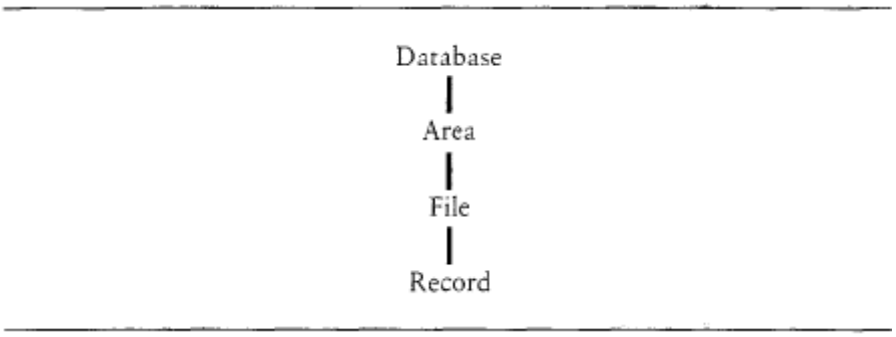

Figura 5 - Exemplo de um *Lock Type Graph* [Bernstein et al., 1987]

Um conjunto de itens de dados estruturado de acordo com um *lock type graph* é chamado de *lock instance graph* (Figura 6).

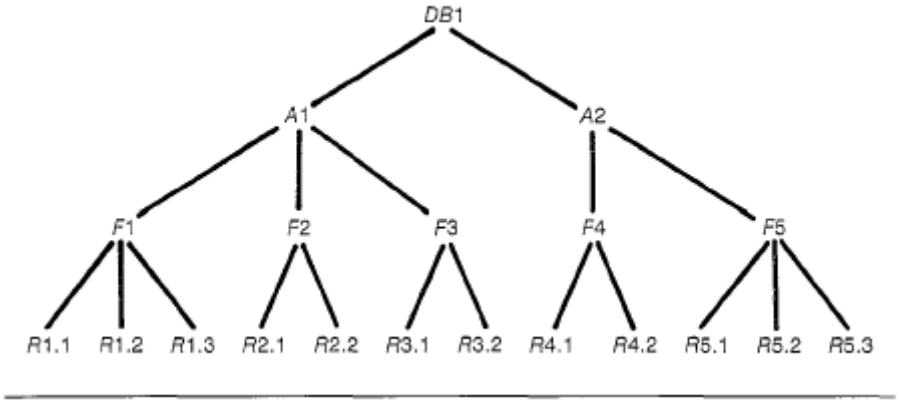

Figura 6 - Exemplo de um *Lock Instance Graph* [Bernstein et al., 1987]

Um bloqueio em um grânulo maior *x* explicitamente bloqueia *x* e implicitamente bloqueia todos os descendentes de *x*, que são grânulos mais finos "contidos" em *x*. Por exemplo, um bloqueio de leitura em uma área implicitamente estabelece bloqueio de leitura nos arquivos e nos registros daquela área.

É necessário também propagar os efeitos de um bloqueio em um grânulo de granularidade fina para os grânulos maiores que o "contém". Em função disto, cada tipo de bloqueio tem um tipo de bloqueio de intenção associado. Portanto, além dos bloqueios de escrita e leitura, no MGL, há também bloqueio de leitura e escrita de intenção. Antes de estabelecer um bloqueio em *x*, é necessário garantir que não haja bloqueio nos ancestrais de *x* que implicitamente bloqueiem *x* em um modo conflitante. Para garantir isto, são estabelecidos bloqueios de intenção nos ancestrais de *x*. Por exemplo, antes de estabelecer um bloqueio de leitura *rl[x]* no registro x, são estabelecidos bloqueios de leitura de intenção *ir* nos

seguintes ancestrais de *x*: banco de dados, área e arquivo (nesta ordem). Para qualquer *y*, *irl[y]* e *wl[y*] conflitam. Consequentemente, com *irl[y]* em todos os ancestrais *y* de *x*, jamais haverá um *wl[y]* que implicitamente bloqueie *x* em modo de escrita. Pela mesma razão, *iwl[x]* conflita com *rl[x]* e *wl[x]* (Figura 7). Na Figura 7 é apresentada a matriz de compatibilidade entre todos os tipos de bloqueio, incluindo os bloqueios de intenção. Nesta matriz existe um tipo de bloqueio composto, *riw*. Um *riwl[x]* é logicamente o mesmo de estabelecer, ao mesmo tempo, os bloqueios *rl[x]* e *iw[x]*. Este bloqueio composto é útil, por exemplo, quando se tem uma transação que lê todos os registros de um arquivo e escreve em apenas alguns. Esta transação precisa, portanto, estabelecer ao mesmo tempo no arquivo um bloqueio de leitura (para ler todos os registros) e um bloqueio de escrita de intenção (para escrever apenas alguns registros).

	r	*w*	*ir*	*iw*	*riw*
r	y	n	y	n	n
w	n	n	n	n	n
ir	y	n	y	y	y
iw	n	n	y	y	n
riw	n	n	y	n	n

Figura 7 - Matriz de compatibilidade do MGL [Bernstein et al., 1987]

Finalizando, a seguir são apresentados os passos do protocolo MGL para os casos onde o *lock instance graph G* é uma árvore[4]. Sendo *G* uma árvore, cada transação T_i adquire e libera bloqueios de acordo com o seguinte protocolo:

1. Se *x* não for a raiz de G, então para estabelecer $rl_i[x]$ ou $irl_i[x]$, T_i tem que ter um bloqueio *ir* ou *iw* no pai de x.
2. Se *x* não for a raiz de G, então para estabelecer $wl_i[x]$ ou $iwl_i[x]$, T_i tem que ter um bloqueio *iw* no pai de x.
3. Para ler (ou escrever) *x*, T_i tem que ter um bloqueio *r* ou *w* (ou *w*) em algum ancestral[5] de *x*. Um bloqueio em *x* é um bloqueio

[4] Árvore - Grafo acíclico, onde dado quaisquer par de nós (i, j) existe somente um único caminho entre i e j.

[5] Um nó é considerado ancestral de si mesmo. Na teoria de grafos, um ancestral diferente de si mesmo é chamado de ancestral próprio (em inglês, "*proper ancestor*").

explícito em *x*; um bloqueio em um ancestral de *x* é um bloqueio implícito em *x*.

4. Uma transação não pode liberar um bloqueio de intenção em um item de dado *x*, se ela possuir bloqueio em algum filho de *x*.

Regras 1 e 2 implicam que para estabelecer um bloqueio em *x* tem que primeiro estabelecer os bloqueios de intenção apropriados em todos os ancestrais de *x*. A regra 3 implica que bloqueando *x* implicitamente bloqueia todos os descendentes de *x*. E a regra 4 diz que bloqueios devem ser liberados na ordem da folha para a raiz, que é o inverso da ordem da raiz para a folha na qual foram adquiridos.

Apesar de ter sido apresentado o protocolo para o caso específico onde *G* é uma árvore, a ideia pode ser perfeitamente estendida para grafos, como será descrito no capítulo 3, que descreve o modelo de bloqueio aqui proposto.

Níveis de Isolamento

Uma das maiores causas de degradação de desempenho são consultas, ou seja, requisições apenas de leitura para suporte a decisão e relatórios. Tipicamente, transações de leitura são mais demoradas do que transações de escrita e acessam uma grande quantidade de itens de dados. Portanto, se transações de leitura usarem o protocolo 2PL, elas, frequentemente, estabelecerão muitos bloqueios e manterão estes bloqueios por um longo período. Isto causa um longo, muitas vezes intolerável, atraso das transações de escrita [Bernstein e Newcomer, 2009].

Diante deste problema, muitos sistemas relaxam a serialização para consultas. Num sistema como estes, que é o caso da maioria dos SGBD[6]s comerciais, para os acessos de leitura, podem ser utilizadas regras mais fracas de bloqueio do que as previstas no protocolo 2PL padrão. Este relaxamento do protocolo 2PL para leitura deu origem aos chamados níveis de isolamento. Tabela 1 e Tabela 2 mostram, respectivamente, o que significam os níveis de isolamento com relação aos problemas de leitura e à aquisição/duração de bloqueios.

[6] SGBD - Sistema de Gerenciamento de Banco de Dados

Tabela 1 - Níveis de isolamento e problemas de leitura

Nível de Isolamento	Dirty Reads	Non-repeatable Reads	Phantoms Reads
Read Uncommitted	Podem ocorrer.	Podem ocorrer.	Podem ocorrer.
Read Committed	Não ocorrem.	Podem ocorrer.	Podem ocorrer.
Repeatable Read	Não ocorrem.	Não ocorrem.	Podem ocorrer.
Serializable	Não ocorrem.	Não ocorrem.	Não ocorrem.

Tabela 2 - Níveis de isolamento e bloqueios

Nível de Isolamento	Write *Lock*	Read *Lock*	Range *Lock*
Read Uncommitted	Não tem.	Não tem.	Não tem.
Read Committed	Até o fim da transação.	Só durante a consulta ("Select")	Não tem.
Repeatable Read	Até o fim da transação.	Até o fim da transação.	Não tem.
Serializable	Até o fim da transação.	Até o fim da transação.	Até o fim da transação.

Em termos de banco de dados relacionais, em geral, o grânulo de bloqueio é a tupla da relação. E no caso de *Range Lock*, parte ou toda a relação é bloqueada.

Existe ainda outro nível de isolamento chamado *Snapshot* que será abordado na seção seguinte.

Multiversão

Uma boa técnica para garantir que transações de leitura leiam dados consistentes sem impactar na execução de transações de escrita é uso de múltiplas versões. Nesta técnica, atualizações não sobrescrevem cópias existentes de itens de dados. Em vez disso, quando uma transação de escrita modifica um item de dados existente, é criada uma nova cópia do item de dado, uma nova versão. Portanto, cada item de dado consiste de uma sequência de versões, uma versão para cada atualização ocorrida. Por exemplo, na Figura 8, um item de dado é uma linha da tabela e cada versão é uma linha separada.

Transaction Identifier	Previous *Transaction*	*Employee Number*	*Name*	Department	Salary
174	null	3	Tom	Hat	$20,000
21156	174	3	Tom	Toy	$20,000
21153	21156	3	Tom	Toy	$24,000
21687	null	43	Dick	Finance	$40,000
10899	null	19	Harry	Appliance	$27,000
21687	10899	19	Harry	Computer	$42,000

Figura 8 - Exemplo de múltiplas versões [Bernstein e Newcomer, 2009]

Para distinguir entre diferentes versões do mesmo item de dado, cada versão contém o identificador único da transação que a gerou, bem como aponta para a versão anterior. Além disso, é mantida uma lista de IDs das transações finalizadas com sucesso (*commit list*).

O uso de múltiplas versões tem por objetivo evitar o uso de bloqueios de leitura por meio de um nível de isolamento chamada *Snapshot*. Quando uma transação de leitura executa neste nível de isolamento, o gerenciador do banco de dados primeiro lê o estado corrente da *commit list* e associa este estado à transação durante toda a sua execução. Quando a transação de leitura pede por um item de dados x, o gerente do banco de dados seleciona a última versão de x gerada por uma transação presente na *commit list* associada à transação de leitura. Não há a necessidade de bloqueios de leitura, pois não há atualização de uma versão existente, apenas criação de novas versões.

No nível de isolamento Snapshot, a transação lê um estado consistente que existe no momento que ela inicia. Qualquer modificação depois deste momento será realizada por transações que não se encontram na *commit list* da transação de leitura, portanto será ignorada. Apesar do estado lido ser consistente, este vai se tornando obsoleto com o passar do tempo, enquanto a transação está em curso [Bernstein e Newcomer, 2009].

Desempenho

Bloqueios afetam o desempenho de maneira significativa. Há duas políticas para lidar quando ocorre um conflito: bloquear ou reiniciar. Ambas têm seu impacto considerável no *throughput* (número de transações concluídas por unidade de tempo) do sistema.

Como ilustra a Figura 9, até atingir um ponto de saturação, a aquisição de bloqueios introduz um retardo modesto, significante, mas nada sério. Todavia, ao atingir certo ponto de saturação, quando um número muito alto de transações requisitam bloqueios, repentinamente, muitas transações começam a ficar bloqueadas (ou reiniciar, se for o caso) e poucas transações conseguem progredir. O *troughput* deixa de crescer. Surpreendentemente, se novas transações continuarem chegando, o *throughput* na verdade até decresce. Este fenômeno é chamado de *lock thrashing* [Bernstein e Newcomer, 2009].

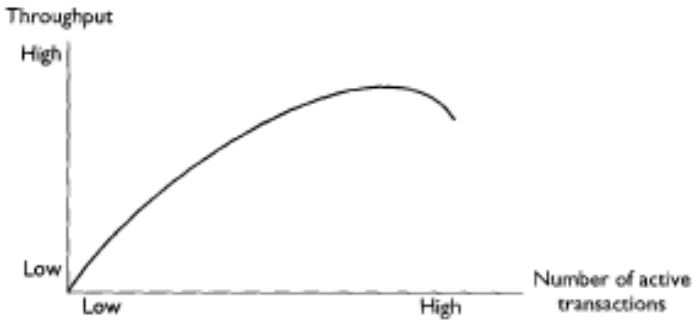

Figura 9 - *Lock Thrashing* [Bernstein e Newcomer, 2009]

Thrashing tem duas componentes: contenção de recursos (*RC-Thrashing*) e contenção de dados (*DC-Thrashing*). *RC-Thrashing* ocorre com relação ao uso da memória, processador, entrada e saída, etc. Já *DC-Thrashing* diz respeito apenas às filas que se formam para acesso a dados, devido ao controle de concorrência.

Em [Bernstein et al., 1987] é apresentado um modelo matemático que permite medir a contenção de dados por meio da carga do sistema: $W = k^2 N / D$, onde k é o número de bloqueios que uma transação requisita, D é o número de itens de dados, e N é o número de transações ativas (Figura 10). *DC-Thrashing* ocorre por volta de W = 1.5 (valor obtido a partir de um modelo de desempenho e confirmado por simulações).

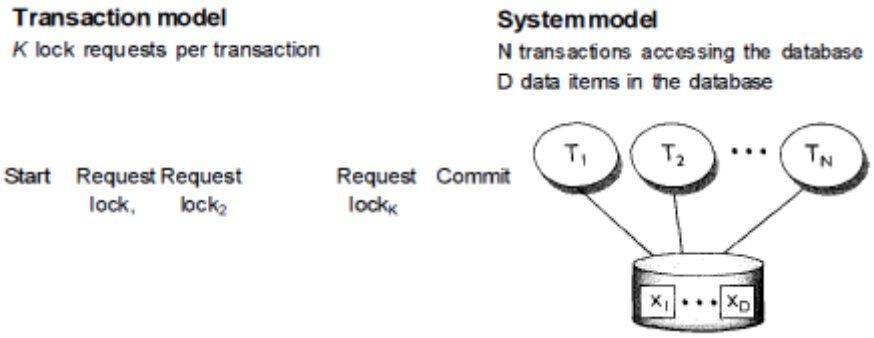

Figura 10 - Modelo matemático de transações [Bernstein e Newcomer, 2009]

Outro fator que tem impacto significante no desempenho é a granularidade do bloqueio, variável D no modelo matemático. Três fatores determinam o efeito da granularidade no desempenho [Bernstein et al., 1987]:

- ***Overhead***

  Quanto mais fina a granularidade, mais bloqueios terão que ser adquiridos, incorrendo em um maior *overhead*.

- **Contenção de Dados**

  Intuitivamente, quanto mais fina a granularidade, maior será a concorrência e, portanto, melhor o desempenho. Na verdade, está intuição não está totalmente correta. Granularidade mais fina reduz a probabilidade de conflitos *por requisição*. Não obstante, mais bloqueios são necessários, consequentemente, o número de conflitos que uma transação se depara pode aumentar. Isto pode ser visto no modelo matemático anterior ($W = k^2 N / D$). Se um aumento em D causa um aumento proporcional em k, então a contenção de dados (W) aumenta.

- **Contenção de Recursos**

  Perceba que contenção de dados alivia contenção de recursos, pois transações "dormem" na fila por bloqueios. Refinando a granularidade pode, consequentemente, liberar mais transações que irão consumir mais recursos.

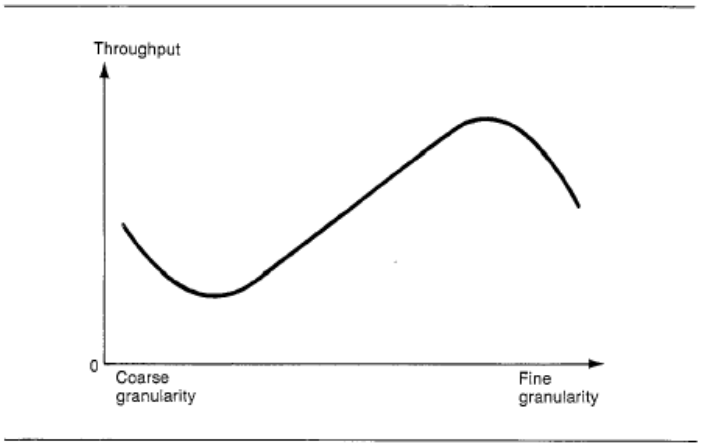

Figura 11 - Curva geral de granularidade [Bernstein et al., 1987]

Estes três fatores combinados geram a curva apresentada na Figura 11. O declínio inicial no throughput é causado pelo aumento em *k* quando *D* aumenta, gerando um aumento no *overhead* de bloqueio e na contenção de dados. À medida que a granularidade vai ficando mais fina, o número de bloqueios que a transação requisita se aproxima do máximo de um bloqueio para cada acesso. Agora k se torna insensível a D, com isso W diminui, e o *throughput* aumenta se a granularidade continuar diminuindo. O declive final é causado pela contenção de recursos. Supondo que haja um número suficiente de transações no sistema para causar *RC-Thrashing*, então refinando a granularidade reduz a contenção de dados, desbloqueando as transações, e causando o decaimento na curva por *RC-Thrashing*.

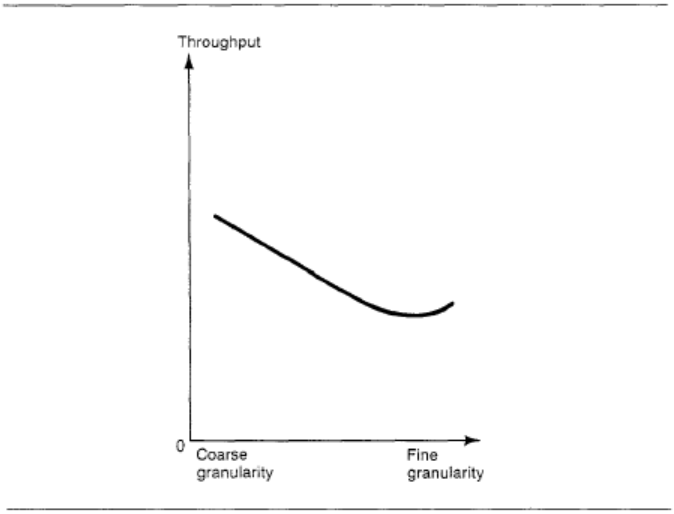

Figura 12 - Curva de granularidade - transações longas [Bernstein et al., 1987]

Em certos sistemas a curva geral da Figura 11 pode não ocorrer por inteiro. Para transações longas, que acessam uma porção significante do banco de dados, mesmo a granularidade mais fina possível não fará o *troughput* crescer após o declínio inicial, como na Figura 12. Cada transação deve, portanto, estabelecer um bloqueio em um grânulo mais grosso (o banco de dados inteiro, em certos casos). Para transações curtas, *k* pode rapidamente ficar insensível a *D*, de forma que o declínio inicial seja minimal. E ainda, se número N de transações não for excessivo, o declínio final na curva de granularidade não ocorrerá, como mostra a Figura 13.

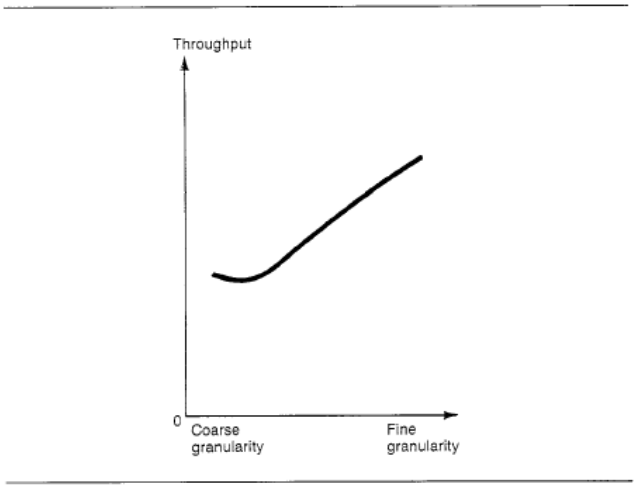

Figura 13 - Curva de granularidade - transações curtas [Bernstein et al., 1987]

2.1.2.2. Bloqueio Otimista

A abordagem pessimista apresentada na seção anterior assume um ambiente transacional onde conflitos são eventos frequentes e, portanto, são necessárias medidas de *prevenção*. Partindo da premissa oposta de que conflitos são eventos raros, uma abordagem menos restritiva, de *detecção*, pode ser empregada. Esta abordagem é comumente chamada de bloqueio otimista. Vale ressaltar que, a despeito do nome, não existe estabelecimento de bloqueio algum.

Bloqueio otimista essencialmente permite que operações simplesmente passem, mas verifica no momento propício se o escalonamento produzido permanece serializável ou não. Em outras palavras, ocasionalmente a saída produzida precisa ser validada, o que lhe confere o nome de protocolo de validação (às vezes também chamado de *certificador*) [Weikum e Vossen, 2002].

Conforme descrito em [Weikum e Vossen, 2002], no bloqueio otimista, a execução de uma transação se dá em três fases:

1. **Fase de Leitura**

   A transação é executada, porém com todas as alterações aplicadas em uma área de trabalho privada, específica da transação. Portanto, as "versões" privadas dos itens de dados alterados pela transação não são visíveis para outras transações (ainda). Perceba que ao final desta fase da transação *t*, o conjunto de itens lidos RS(*t*) e o conjunto de itens modificados WS(*t*) são conhecidos.

2. **Fase de Validação**

   Nesta fase, também conhecida por *pré-commit*, é obtido o bloqueio otimista, indicando que não houve conflito e que os dados podem ser transferidos da área de trabalho privada para o banco de dados. Obter o bloqueio otimista significa simplesmente validar uma transação que está pronta para finalizar (*commit*). Ou seja, verificar se sua execução está de acordo com um escalonamento serializável, se não houve conflito, podendo então copiar seus dados no banco de dados. Se houver conflito a transação é abortada; caso contrário começa a próxima fase.

   Essencialmente há duas estratégias de bloqueio otimista:

   - *backward-oriented optimistic concurrency control* (BOCC) - é realizado um teste de conflito com todas as transações que já foram aceitas (*committed*).
   - *forward-oriented optimistic concurrency control* (FOCC) - é realizado um teste de conflito com todas as transações que estão executando em paralelo, mas que ainda estão na fase de leitura.

3. **Fase de Escrita**

   O conteúdo da área de trabalho da transação é transferido para o banco de dados, concluindo o *commit*.

É assumido que as duas últimas fases são executadas atomicamente, como uma "região crítica" ininterrompível, significando que todas as outras transações são suspensas durante esta região crítica.

2.2. Web Semântica

A Web corrente é inteiramente voltada para ser consumida por humanos, ou seja, os dados nela armazenados não são estruturados, sendo puramente orientados a exibição. Em outras palavras, a Web original foi construída de tal forma que ela não se preocupa com o significado da informação provida pelos sites. Na Web atual, os computadores conseguem apenas apresentar as informações contidas nos documentos, porém não as compreende.

Imagine se a Web fosse concebida de tal forma que o significado de cada documento pudesse ser obtido a partir de uma coleção de declarações (*statements*) e as máquinas fossem capazes de entender estas declarações, tomando decisões "inteligentes" para nos auxiliar. Por exemplo, máquinas de busca filtrarem os conteúdos ou até mesmo nos dar a resposta, agentes de software automaticamente fazerem integração de informação a partir de vários sites ou, ir além, fazendo *Web data mining*, chegando a resultados preciosos.

Será que é possível reconstruir ou estender a Web, adicionando estas declarações aos documentos, de forma que as máquinas possam utilizar esta informação extra para entenderem o significado de um dado documento?

A resposta é sim e esta nova Web é denominada Web semântica. A Web semântica provê as tecnologias e padrões necessários para tornar possível adição de significado inteligível por máquina à Web atual, de forma que computadores possam automaticamente executar tarefas até então executadas manualmente, em larga escala [Yu, 2011].

2.2.1. Definição e Requisitos

O termo "*Semantic Web*" foi originalmente criado pelo diretor do World Wide Web Consortium (W3C)[7], Tim Bernes-Lee e formalmente apresentado ao mundo em Maio de 2011, em seu artigo "*The Semantic Web*", na revista *Scientific American* [Bernes-Lee et al., 2001]:

> "*The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*"

[7] W3C - http://www.w3.org

Para o *W3C Semantic Web Activity*[8], grupo de pesquisa e padronização da Web semântica no W3C, a Web semântica tem o seguinte significado:

> "*The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.*"

Em [Yu, 2011], a Web semântica é definida como uma coleção de tecnologias e padrões que permitem máquinas entenderem o significado (semântica) da informação disponibilizada na Web.

A Tabela 3 enumera os requisitos da Web semântica e como estes são mapeados para as tecnologias e padrões.

Tabela 3 - Mapeamento dos requisitos da Web semântica em padrões

Requisitos	Tecnologias e Padrões
Tem que haver um modelo para representar o conhecimento na Web e este modelo tem que ser facilmente processado (entendido) por máquinas.	*Resource Description Framework*[9] *(RDF)*
Este modelo tem que ser aceito como um padrão por todos os Web sites; assim *statements* contidos em diferentes sites, serão todos similares.	*Resource Description Framework (RDF)*
Tem que haver uma forma de adicionar estes *statements* em todos os Web sites, manualmente ou automaticamente.	*Semantic markup, RDFa*[10]*, Microformats*[11]*, GRDDL*[12]

[8] W3C Semantic Web Activity - http://www.w3.org/2001/sw/

[9] RDF - http://www.w3.org/2001/sw/RDFCore/

[10] RDFa - http://www.w3.org/TR/xhtml-rdfa-primer/

[11] Microformats - http://microformats.org

[12] GRDDL - http://www.w3.org/2001/sw/grddl-wg/

Requisitos	Tecnologias e Padrões
Os *statements* contidos em diferentes Web sites não podem ser totalmente arbitrários. Eles devem ser criados usando termos e relacionamentos comuns, isto é, um vocabulário comum para um dado domínio.	*Ontologias Específicas de Domínio / Vocabulários*
Tem que haver uma linguagem comum para criação de vocabulários e ontologias.	*RDF Schema*[13] *(RDFS),* *Web Ontology Language*[14] *(OWL)*
Os agentes de software devem ser capazes de processar (entender) cada *statement* que eles coletem, com base nos termos e relacionamentos comuns usados para criar os *statements*.	*Ferramentas e Frameworks para processamento de ontologias*
Os agentes de software devem ser capazes de "raciocinar", gerando novos *statements*, com base no seu entendimento dos termos e relacionamentos comuns.	*Inferência* (em inglês, *Reasoning*) com base em ontologias
Os agentes de software devem ser capazes de submeter consultas sobre os *statements* coletados.	*SPARQL*[15]
Os agentes de software devem ser capazes de inserir e remover *statements* nas bases de conhecimento.	*SPARUL*[16] *(SPARQL/Update)*

[13] RDFS - http://www.w3.org/TR/rdf-schema/

[14] OWL - http://www.w3.org/TR/owl-ref/

[15] SPARQL - http://www.w3.org/TR/rdf-sparql-query/

[16] SPARUL - http://www.w3.org/Submission/SPARQL-Update/

2.2.2. Modelo RDF

Como visto anteriormente, processamento automatizado dos recursos disponibilizados na Web atual é difícil, uma vez que estes recursos não são compreensíveis por máquinas. *Resource Description Framework* (RDF) [MANOLA e MILLER, 2004] é um padrão criado originalmente em 1999, pelo W3C, para codificação de metadados para descrever, de forma semi-estruturada, recursos disponibilizados na Web. Trata-se de uma linguagem de propósito geral, independente de domínio, voltada para processamento por máquinas, a fim de promover interoperabilidade entre aplicações que trocam informações na Web.

O conceito da Web semântica foi introduzido em 2001, depois do advento do RDF. Todavia, dado que o objetivo da Web semântica é tornar a Web inteligível por máquinas e que conhecimento representado usando RDF é estruturado, isto é, compreensível por máquinas, o RDF foi escolhido como modelo de representação de conhecimento na Web semântica vislumbrada pelo W3C. Podemos dizer que o RDF está para a Web semântica assim como o HTML está para a Web tradicional.

No contexto da Web semântica, RDF não é usado apenas para codificar metadados sobre recursos Web, mas, sobretudo, para descrever quaisquer recursos e suas propriedades/relacionamentos existentes no mundo real. Qualquer recurso significa qualquer "coisa" mesmo, concreta ou abstrata, que exista (pessoas, documentos, empresas, eventos, produtos, paradigmas, ideologias, etc.).

O modelo abstrato de RDF é um modelo em grafo constituído de três componentes chave: *statement*, recursos (como sujeito ou objeto) e predicado.

Statement

O objetivo do RDF é dividir a informação (ou conhecimento) em pequenos pedaços, cada pedaço com semântica bem definida de forma que a máquina possa entendê-lo e fazer algo útil com ele. Este pequeno pedaço de conhecimento é chamado *statement* (afirmação, declaração, fato, em português).

Cada *statement* tem a forma de "*sujeito-predicado-objeto*", onde *sujeito* e *objeto* são nomes (únicos) de duas "coisas" do mundo real, e *predicado* é nome (também único) de um relacionamento que conecta estas duas coisas. No contexto de RDF, a "coisa" denotada por um dado sujeito ou objeto é chamada

recurso. Como um *statement* RDF é composto por três componentes, ele também é comumente chamado de *tripla*. A Figura 14 apresenta o grafo de um *statement* RDF. Trata-se de um digrafo (grafo direcionado), onde o sujeito e o objeto são nós e o predicado é aresta que aponta do sujeito para o objeto.

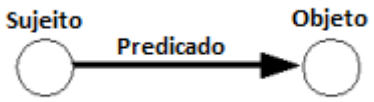

Figura 14 - Um *statement* RDF

Resumindo, cada *statement* representa um fato único; um conjunto de *statements* representa uma dada porção de informação ou conhecimento; e um conjunto de *statements* é chamado de (sub)grafo RDF.

Recurso

Conforme dito anteriormente um recurso é qualquer "coisa" descrita por *statements* RDF, denotada por um dado sujeito ou objeto. Em RDF, todo recurso tem um nome global único no mundo, permitindo que *statements* sobre este recurso sejam publicados na Web, por qualquer um, em qualquer lugar e a qualquer momento, sem possibilidade de confusão.

A arquitetura da Web tradicional já provê um identificador único, a *Uniforme Resource Locator* (URL). A URL representa o endereço de rede do recurso Web. No entanto, é necessário um mecanismo de identificação mais geral que sirva para identificar qualquer recurso, inclusive recursos que não fazem parte fisicamente da Web e, portanto, não possuem um endereço de rede. Por exemplo, eu, uma pessoa, posso ser identificado na Web, mas certamente não faço parte da Web e, consequentemente, não posso ser recuperado a partir da Web. Felizmente a Web provê uma forma mais geral de identificação para estes propósitos, chamada *Uniform Resource Identifier* (URI). URLs são um tipo especial de URI. Diferentemente de URLs, URIs não se limitam a identificar recursos que possuam um endereço de rede na Web. De fato, uma URI pode ser usada para identificar qualquer coisa que precise ser referenciada em um *statement* RDF.

Concluindo, no mundo RDF, todo recurso é identificado por um nome global único que é uma URI.

Um recomendação muito importante é que se já houver uma URI que identifique o recurso, devemos, sempre que possível, reusá-la, ao invés de criar uma outra URI para representar o mesmo recurso. A razão é que se todos os Web sites utilizarem a mesma URI para representar o recurso, fica muito mais fácil integrar todos os *statements*, residentes em cada site, que mencionem o recurso em questão. Caso contrário, ao integrar as informações, é necessário um trabalho extra para identificar que duas ou mais URIs, na verdade, representam o mesmo recurso do mundo real.

Predicado

Num dado *statement* RDF, o predicado denota o relacionamento entre o sujeito e o objeto. O modelo RDF também requer o uso de URIs para nomear predicados, ao invés de uso de simples strings. A primeira razão é similar a razão pela qual devemos usar URIs para nomear sujeitos e objetos, ou seja, evitar ambiguidades, como, por exemplo, relacionamentos homônimos com significados são diferentes. A segunda razão é que o uso de URIs para identificar predicados permite tratar predicados como qualquer outro recurso e, portanto, criar *statements* usando o predicado como sujeito ou objeto. Como será abordado mais a frente, isto é fundamental para a criação de vocabulários e ontologias, onde precisamos definir as propriedades e relacionamentos entre predicados. E por fim, a terceira razão está relacionada ao reuso de predicados pré-existentes que implica num conhecimento compartilhado deste predicados ou conceitos, facilitando a integração de informações por máquinas.

A Figura 15 ilustra um *statement* RDF realista, onde todos os três componentes são nomeados com URIs. Este *statement* RDF corresponde a seguinte declaração em linguagem natural:

> "*O* ***recurso*** *http://www.example.org/index.html possui a* ***propriedade*** *http://purl.org/dc/elements/1.1/creator cujo* ***valor*** *é o recurso http://www.example.org/staffid/85740*"

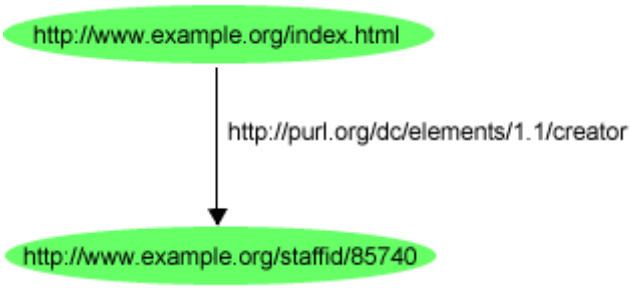

Figura 15 - Um *statement* RDF realista [MANOLA e MILLER, 2004]

Perceba que podemos trocar os nomes dos componentes do *statement* RDF, aproximando-o de nossa linguagem natural, resultando em "*recurso-propriedade-valor*". De fato, um predicado na mais é do que uma propriedade (atributo ou relacionamento) de um recurso. É como é de praxe, toda propriedade tem um valor, que poder ser um valor literal (ou primitivo) ou um outro recurso. A Figura 16 contém três *statements* sobre um mesmo recurso, dois tendo um literal como valor e um tendo outro recurso com valor.

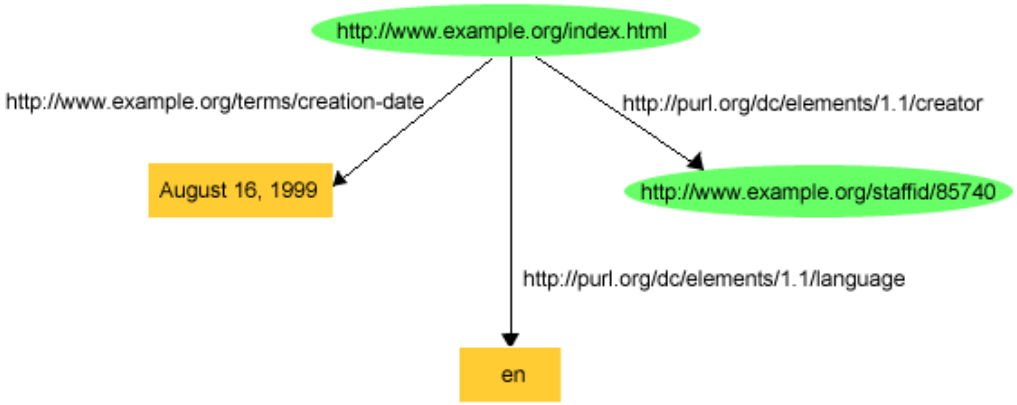

Figura 16 - *Statements* sobre um mesmo recurso [MANOLA e MILLER, 2004]

A Figura 17 e a Figura 18 mostram o mesmo grafo da Figura 16, porém usando a notação em triplas, no lugar da notação gráfica. A Figura 18 usa URIs abreviadas, usando *XML qualified name (QName)*[17], com os seguintes prefixos:

- prefixo `dc:`, *namespace* URI: http://purl.org/dc/elements/1.1/
- prefixo `ex:`, *namespace* URI: http://www.example.org/
- prefixo `exterms:`, *namespace* URI: http://www.example.org/terms/
- prefixo `exstaff:`, *namespace* URI: http://www.example.org/staffid/

[17] *QName* - prefixo de um *URI namespace*, seguido de ":" e um nome local

```
<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/creator> <http://www.example.org/staffid/85740> .
<http://www.example.org/index.html> <http://www.example.org/terms/creation-date> "August 16, 1999" .
<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/language> "en" .
```

Figura 17 - Notação em triplas [MANOLA e MILLER, 2004]

```
ex:index.html  dc:creator            exstaff:85740 .
ex:index.html  exterms:creation-date  "August 16, 1999" .
ex:index.html  dc:language           "en" .
```

Figura 18 - Notação em triplas abreviada [MANOLA e MILLER, 2004]

Existe ainda um tipo especial de nó chamado "nó em branco" ou "nó anônimo" que é um nó que não possui URI, usado frequentemente para modelar valores estruturados mais complexos do que um tipo primitivo ou para reduzir relacionamentos *n*-ários a *n* relacionamentos binários. A Figura 19 mostra o uso de "nó em branco" para representar o valor estruturado da propriedade "http://www.example.org/terms/address" e seus vários campos.

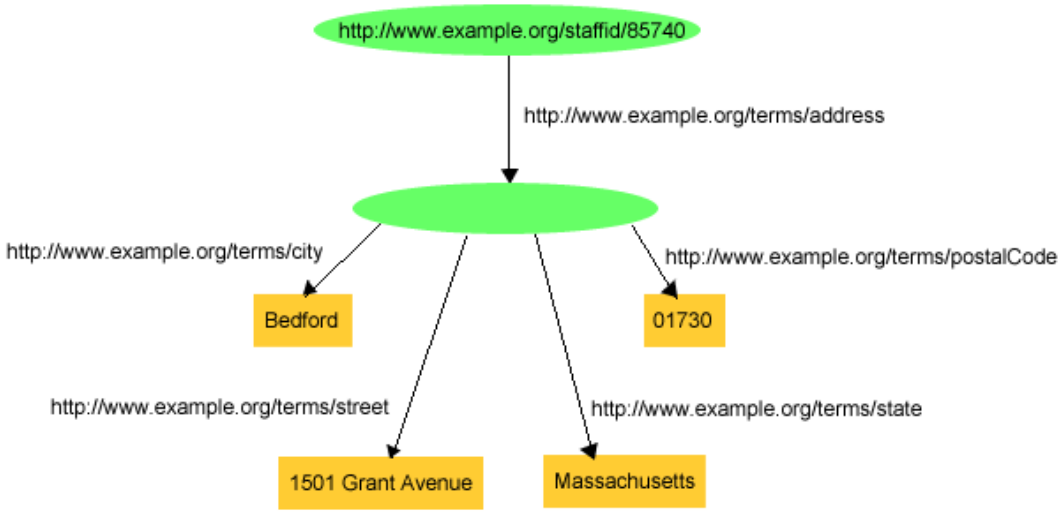

Figura 19 - Uso de "nó em branco" [MANOLA e MILLER, 2004]

Serialização RDF

O modelo RDF descrito até então é um modelo abstrato em grafo para descrever recursos, de forma que máquinas possam processar. O próximo passo é definir alguma sintaxe de serialização para as aplicações criarem e lerem modelos RDF concretos.

O W3C define uma sintaxe XML para este propósito chamada RDF/XML[18]. A Figura 20 apresenta o mesmo grafo da figura 16, serializado em RDF/XML.

[18] RDF/XML - http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210

```
1.  <?xml version="1.0"?>
2.  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3.              xmlns:dc="http://purl.org/dc/elements/1.1/"
4.              xmlns:exterms="http://www.example.org/terms/">

5.    <rdf:Description rdf:about="http://www.example.org/index.html">
6.         <exterms:creation-date>August 16, 1999</exterms:creation-date>
7.         <dc:language>en</dc:language>
8.         <dc:creator rdf:resource="http://www.example.org/staffid/85740"/>
9.    </rdf:Description>

10. </rdf:RDF>
```

Figura 20 - Exemplo de serialização em RDF/XML [MANOLA e MILLER, 2004]

A sintaxe RDF/XML é voltada exclusivamente para máquinas, não sendo nada amigável para leitura humana. De fato existem outras sintaxes de serialização RDF mais amigáveis. São elas, ordenadas da mais complexa para a mais simples: *Notation 3 (N3)*[19], *Turtle*[20] e *N-Triples*[21]. Dentre estas, *Turtle* é bastante popular entre desenvolvedores e foi escolhida como base da sintaxe da linguagem de consulta SPARQL [Prud'hommeaux e Seaborne, 2008].

2.2.3. Vocabulários e Ontologias

Os *statements* contidos em diferentes Web sites não podem ser totalmente arbitrários. Eles devem ser criados usando termos e relacionamentos comuns, isto é, um vocabulário comum para um dado domínio.

Quando descrevemos um recurso, quais as propriedades (predicados) devemos usar para criar os *statements* sobre este recurso? Se cada um de nós inventar suas próprias propriedades, não haverá uma linguagem comum compartilhada. Neste caso, as aplicações não poderão ir além de simplesmente agregar dados RDF distribuídos. Uma linguagem ou vocabulário comum, com classes e propriedades pré-definidas, precisa ser definido, publicado e reusado. Este vocabulário permitirá máquinas fazerem inferências, com base no conhecimento descrito nele.

[19] N3 - http://www.w3.org/DesignIssues/Notation3

[20] Turtle - http://www.w3.org/TeamSubmission/turtle

[21] N-Triples - http://www.w3.org/2001/sw/RDFCore/ntriples

Vocabulário

Uma vez que RDF usa *URIrefs* para nomear "coisas" em *statements*, a comunidade RDF se refere a um conjunto de *URIrefs* (particularmente um conjunto criado para um propósito específico) como um vocabulário. Em geral, as *URIrefs* em um vocabulário são organizadas de tal forma que possam ser representadas por um conjunto de *QNames* usando um mesmo prefixo. Isto é, um namespace *URIref* comum é escolhido para todos os termos em um vocabulário, tipicamente uma *URIref* sob o controle de quem está definindo o vocabulário. *URIrefs* pertencentes ao vocabulário são formadas anexando nome locais ao final da *URIref* comum do vocabulário. Isto forma um conjunto de *URIrefs* com um prefixo comum. Além disso, às vezes o *namespace URIref* do vocabulário é usado como URL de um recurso Web que provê informação extra sobre o vocabulário. Por exemplo, o *prefixo QName* `dc:`, usado nos exemplos anteriores, é associado ao *namespace URIref http://purl.org/dc/elements/1.1/*, referente ao vocabulário Dublin Core. Acessando este *namespace URIref* em um browser, retornará informação adicional sobre o vocabulário Dublin Core (especificamente, um RDF *schema*). Entretanto, isto é apenas uma convenção. RDF não assume que um *namespace URIref* identifica uma recurso Web acessível [MANOLA e MILLER, 2004].

No início desta seção foi dito que um vocabulário permite máquinas fazerem inferências com base no conhecimento descrito nele. Porém, como o conhecimento é codificado em um vocabulário?

Para responder esta pergunta é preciso entender o conceito de ontologia. Com base na definição de vocabulário descrita há pouco, ou seja, um conjunto de termos criado para um propósito específico, podemos entender que vocabulário corresponde aos nomes (*URIrefs*) dados ao conceitos envolvidos. Porém quando nos referimos ao conhecimento envolvendo estes conceitos, ou seja, como estes conceitos estão relacionados entre si, bem como o significado (semântica) de cada um deles, estamos na verdade fazendo alusão à ontologia cujos conceitos foram expressos por meio daquele vocabulário. Os termos ontologia e vocabulário muitas vezes se confundem, sendo comumente utilizados como sinônimos no dia a dia.

A noção de ontologia é oriunda do campo da filosofia, e existem várias definições de ontologia. Uma definição bastante popular é "ontologia é a formalização de uma conceitualização" [Yu, 2011]. No contexto da Web semântica, em [Heflin, 2004] ontologia é definida da seguinte forma:

"*An ontology formally defines a common set of terms that are used to describe and represent a domain.*"
"*An ontology defines the terms used to describe and represent an area of knowledge.*"

Uma ontologia é criada para um domínio específico (educação, música, obras de arte, livros, etc.). Uma ontologia contém conceitos ou classes e relacionamentos entre estas classes. Os relacionamentos entre estas classes podem ser expressos por meio de hierarquias: superclasses e subclasses. Existem ainda outros tipos de relacionamentos expressos por meio de propriedades. Propriedades descrevem várias características e atributos das classes, e também podem ser usadas para definir novas classes, empregando a noção de conjuntos, ou seja, classes como conjuntos de objetos que apresentam certas propriedades, com certos valores.

Tendo os termos e relacionamentos entre estes termos explicitamente definidos, uma ontologia codifica o conhecimento de um dado domínio, de tal forma que este conhecimento possa ser entendido por máquinas, viabilizando a Web semântica. Sendo assim, emerge como requisito fundamental a existência de linguagens que permitam criar ontologias e seus vocabulários, que é o assunto da próxima seção.

2.2.4. RDF Schema e OWL

RDF Schema (RDFS) [Brickley e Guha, 2004] e OWL [Dean e Schreiber, 2004] ambas são linguagens para definição de ontologias. Em outras palavras são um conjunto de termos (vocabulário RDFS e vocabulário OWL) que permitem definir classes, propriedades e seus relacionamentos para um domínio específico de aplicação. Em outras palavras, RDFS e OWL podem ser encarados como meta-vocabulários usados para criar vocabulários. Vocabulários que serão usados para criar documentos RDF distribuídos pela Web.

No entanto, comparado com RDFS, OWL nos provê a capacidade de expressar relacionamentos mais complexos e mais ricos, permitindo, por conseguinte, a criação de aplicações com maior poder de inferência ou raciocínio. OWL, na verdade, é uma extensão de RDFS, com novas construções que oferecem maior expressividade.

RDF Schema

RDF Schema (RDFS) [Brickley e Guha, 2004] é um vocabulário que oferece um conjunto de termos que nos permitem definir classes e propriedades para um dado domínio. Como qualquer outro vocabulário, todos estes termos são identificados por URIs pré-definidas e todas estas URIs compartilham o seguinte prefixo: *http://www.w3.org/2000/01/rdf-schema#*. Por convenção, este prefixo é associado com o *namespace rdfs* (`xmlns:rdfs`), e é tipicamente usado, no formato RDF/XML, com o prefixo rdfs (`<rdfs:Class ...`).

Os termos RDFS podem ser divididos nos seguintes grupos, com base em seus propósitos:

- **classes**
  Termos usados para definir classes. São eles: *rdfs:Resource, rdfs:Class, rdfs:subClassOf*.

- **propriedades**
  Termos usados para definir propriedades. São eles: *rdfs:range, rdfs:domain, rdfs:Literal, rdfs:Datatype, rdfs:subPropertyOf*.

- **miscelânea**
  Termos de utilidade geral. São eles: *rdfs:Label, rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy*.

A Figura 21, a Figura 22 e a Figura 23 exibem, respectivamente, uma hierarquia de veículos na notação gráfica, a mesma hierarquia na notação em triplas e o vocabulário (ontologia) completo de veículos em RDF/XML.

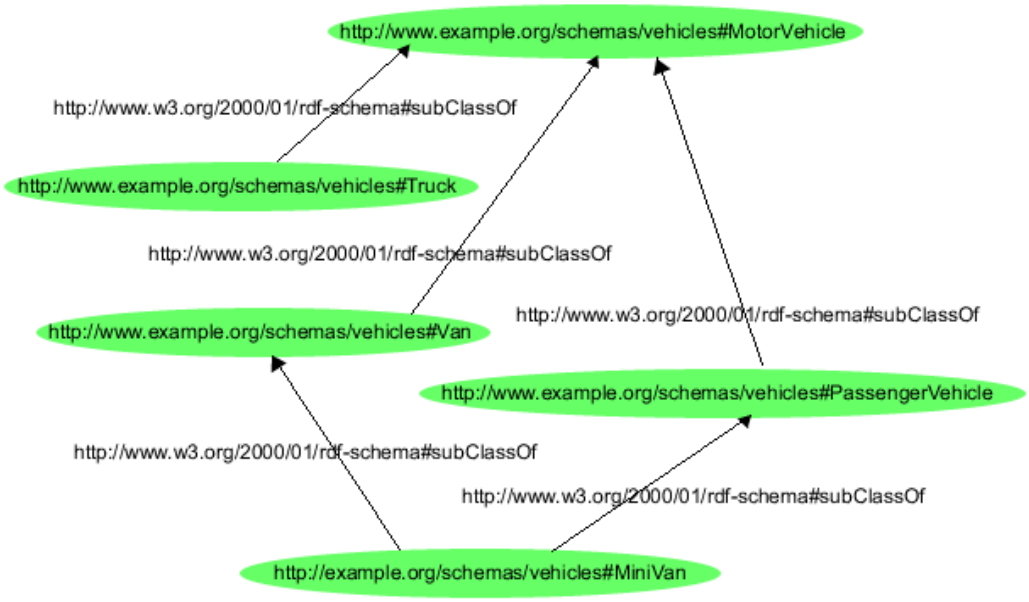

Figura 21 - Hierarquia de classes RDFS (gráfico) [MANOLA e MILLER, 2004]

```
ex:MotorVehicle       rdf:type          rdfs:Class .
ex:PassengerVehicle   rdf:type          rdfs:Class .
ex:Van                rdf:type          rdfs:Class .
ex:Truck              rdf:type          rdfs:Class .
ex:MiniVan            rdf:type          rdfs:Class .

ex:PassengerVehicle   rdfs:subClassOf   ex:MotorVehicle .
ex:Van                rdfs:subClassOf   ex:MotorVehicle .
ex:Truck              rdfs:subClassOf   ex:MotorVehicle .

ex:MiniVan            rdfs:subClassOf   ex:Van .
ex:MiniVan            rdfs:subClassOf   ex:PassengerVehicle .
```

Figura 22 - Hierarquia de classe RDFS (triplas) [MANOLA e MILLER, 2004]

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://example.org/schemas/vehicles">

<rdfs:Class rdf:ID="MotorVehicle"/>

<rdfs:Class rdf:ID="PassengerVehicle">
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Truck">
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Van">
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="MiniVan">
  <rdfs:subClassOf rdf:resource="#Van"/>
  <rdfs:subClassOf rdf:resource="#PassengerVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Person"/>

<rdfs:Datatype rdf:about="&xsd;integer"/>

<rdf:Property rdf:ID="registeredTo">
  <rdfs:domain rdf:resource="#MotorVehicle"/>
  <rdfs:range rdf:resource="#Person"/>
</rdf:Property>

<rdf:Property rdf:ID="rearSeatLegRoom">
  <rdfs:domain rdf:resource="#PassengerVehicle"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>

<rdf:Property rdf:ID="driver">
  <rdfs:domain rdf:resource="#MotorVehicle"/>
</rdf:Property>

<rdf:Property rdf:ID="primaryDriver">
  <rdfs:subPropertyOf rdf:resource="#driver"/>
</rdf:Property>

</rdf:RDF>
```

Figura 23 - Vocabulário de veículos em RDFS [MANOLA e MILLER, 2004]

OWL

O propósito da Web Ontology Language OWL [Dean e Schreiber, 2004] é o mesmo do RDFS, ou seja, definir ontologias. No entanto, OWL é atualmente a linguagem mais utilizada porque oferece tudo que existe no RDFS e mais. OWL é uma extensão do RDFS, derivada da linguagem de ontologias para Web DAML+OIL [Harmelen, 2001].

Em OWL, todas as URIs compartilham o seguinte prefixo: *http://www.w3.org/2002/07/owl#*, e por convenção, este prefixo é associado com

o *namespace owl* (`xmlns:owl`), e é tipicamente usado, no formato RDF/XML, com o prefixo owl (`<owl:Class ...`).

Um *statement* básico em OWL é denominado axioma. Uma ontologia OWL pode ser vista como um conjunto de axiomas. Além disso, esta ontologia afirma que estes axiomas são verdadeiros. Dado que um axioma é um *statement*, obviamente ele envolve alguma classe, propriedade e, às vezes, algum indivíduo (instância). Estas classes, propriedades e indivíduos podem ser vistos como constituintes atômicos de axiomas e são comumente chamados de entidades. No contexto de OWL, indivíduo também é chamado de objeto, categoria é sinônimo de classe e propriedade também é conhecida por relação [Yu, 2011].

OWL permite combinar diferentes classes e/ou propriedades para criar novas classes e propriedades. Estas combinações podem ser bastante complexas e são chamadas de expressões. Expressões são a principal razão do poder de expressividade de OWL.

OWL também prevê um mecanismo de extensão de URI chamado *Internationalized Resource Identifier*[22] (IRI) que tem o mesmo propósito de URI, porém faz uso de todos os caracteres Unicode. URIs são limitadas ao caracteres ASCII. O IRI foi criado para atender línguas que fazem uso de caracteres especiais, como japonês.

OWL oferece aspectos interessantes. Dentre eles, temos:

- Definição de classes anônimas por meio da construção *owl:Restriction* que adiciona alguma restrição em alguma propriedade. Há dois tipos de restrição: restrição de cardinalidade e restrição de valor.
- Definição de classes usando operadores de conjunto (união e interseção).
- Definição de classes por meio de enumeração (de instâncias), equivalência e disjunção.
- Dois tipos de propriedade: *owl:ObjectProperty* e *owl:DatatypeProperty*. A primeira usada para conectar um recurso a outro recurso e a segunda para conectar um recurso a um valor literal (*rdfs:Literal ou XML schema datatype*).
- Vários modificadores de propriedade. Propriedade simétricas, transitiva, funcional, inversa, funcional-inversa.

[22] IRI - http://www.w3.org/International/articles/idn-and-iri/

A última recomendação do W3C é a OWL 2[23]. As novas características podem ser classificadas em cinco categorias:

- Açúcar sintático para facilitar a construção de alguns *statements*.
- Novas construções para aumentar a expressividade. Por exemplo, novas propriedades como propriedade reflexiva, irreflexiva e assimétrica.
- Suporte para novos *datatypes*. A linguagem oferece novos *datatypes* e permite o projetista criar os seus próprios *datatypes*.
- Funcionalidade simples de metamodelagem e novos aspectos de anotação. A funcionalidade de metamodelagem inclui uma nova característica chamada *punning* que permite usar a mesma IRI para nomear um indivíduo e uma classe. Quanto as anotações, pode-se adicionar anotações para axiomas, adicionar informação de *domain* e *range* para propriedade de anotação, adicionar anotações em anotações, etc.
- Novas sublinguagens ou *profiles*: OWL 2 EL, OWL 2 QL e OWL 2 PL. Esta sublinguagens oferecem diferentes níveis do *trade-off* entre expressividade e eficiência.

2.2.5. SPARQL

SPARQL é um acrônimo recursivo para *SPARQL Protocol and RDF Query Language*. Portanto, SPARQL é um linguagem de consulta e um protocolo de acesso para dados RDF. Podemos dizer que SPARQL está para dados RDF assim como SQL está para dados relacionais.

A recomendação SPARQL do W3C consiste em três especificações:

- *SPARQL Query Language specification* [Prud'hommeaux e Seaborne, 2008]
  Descreve as construções da linguagem de consulta para RDF.

[23] OWL 2 - http://www.w3.org/TR/owl2-overview/#ack

- *SPARQL XML Results Format specification* [Becket e Broekstra, 2008]
  Define um formato XML para serialização dos resultados das consultas.

- *SPARQL Protocol for RDF specification* [Clark et al., 2008]
  Usa WSDL[24] 2.0 para definir protocolos HTTP e SOAP[25] para consultas a banco de dados RDF remotos.

Como era de se esperar, os dados RDF não residem apenas serializados em documentos estáticos. Com o advento de RDF, em especial associado ao movimento da Web semântica, começaram a surgir os bancos de dados RDF, também conhecidos como *RDF Data Store* ou *Triple Store*.

Bancos de dados RDF são similares aos bancos de dados tradicionais, onde podemos armazenar *RDF statements* (ou triplas) e recuperá-los depois usando uma linguagem de consulta (em geral, SPARQL). No entanto, banco de dados RDF têm a vantagem de saber de antemão como será o esquema de armazenamento, pois todos os registros terão sempre a mesma forma *sujeito-predicado-objeto*, ou seja, sempre triplas e, portanto, os banco de dados RDF podem ser otimizados para este esquema de triplas. Um banco de dados RDF pode ser projetado "do zero" ou pode ser criado em cima de um banco de dados relacional tradicional. A Tabela 4 apresenta alguns bancos de dados RDF.

[24] WSDL 2.0 - http://www.w3.org/TR/wsdl20-primer/

[25] SOAP 1.2 - http://www.w3.org/TR/soap12-part1/

Tabela 4 - Alguns RDF *Stores*

Banco de Dados RDF	***Web Site***
4store	http://www.4store.org
AllegroGraph	http://www.franz.com/agraph/allegrograph/
ARC	http://arc.semsol.org
Bigdata	http://www.bigdata.com/blog/
Joseki	http://www.joseki.org
Mulgara	http://www.mulgara.org
OWLIM	http://www.ontotext.com/owlim
Redland	http://librdf.org
Sesame	http://www.openrdf.org
Virtuoso	http://virtuoso.openlinksw.com

Consultas SPARQL são realizadas em um banco de dados RDF por meio de um *SPARQL endpoint*. Um *SPARQL endpoint* é uma interface que usuários (humanos ou aplicações) podem acessar para realizar consultas SPARQL. Para humanos, este *endpoint* pode ser uma aplicação *stand-alone* ou Web. Para aplicações, o *endpoint* é um conjunto de APIs usadas para acessar os dados. Um SPARQL endpoint pode ser configurado para retornar os resultados em vários formatos (RDF/XML, Turtle, HTML para humanos, etc.).

Sob o ponto de vista de linguagem de consulta, SPARQL oferece as seguintes formas de consulta:

- **SELECT** *query*

  A forma mais comum, similar ao SELECT de SQL, usada para retornar item de dados (*variable bindings*), ou seja, operação de projeção. Por isso, esta forma de consulta também é conhecida por *Projection query*.

- **ASK** *query*

  Consulta booleana, ou seja, consulta cujo retorno é verdadeiro ou falso, usada para tomada de decisão. Perguntas, como por exemplo, "*Zico foi um jogador de futebol?*".

- **DESCRIBE** *query*

  Usada quando não se sabe muito coisa sobre o grafo que se quer pesquisar. Neste caso, pedimos ao processador de consultas para "descrever" um recurso *x*, e ele retorna alguma informação sobre o recurso. Qual informação retornar fica a cargo do processador de consultas.

- **CONSTRUCT** *query*

  Usada para construir um novo grafo com base nas triplas retornadas. Muito útil para especificar regras do tipo "quando vir isto, conclua aquilo", ou seja, gerar novos *statements*. A regra de lógica "*Se Mark é um homem, e todos os homens são mortais, então Mark é mortal*", por exemplo.

Todos estes tipos de consulta são baseados em dois conceitos básicos: *triple pattern* e *graph pattern*.

Similar a uma tripla RDF, um SPARQL *triple pattern* também possui os três componentes *sujeito-predicado-objeto*. A diferença que um *triple pattern* pode incluir variáveis. Qualquer um dois três componentes pode ser uma variável em um *triple pattern*. Uma variável é sempre prefixada com o caractere ? (ou $) e funciona com espécie de *placeholder* que pode assumir qualquer valor, ou seja, os valores das triplas do banco de dados que "encaixam" no padrão definido pelo *triple pattern*. Já um conjunto de *triple patterns* formam um *graph pattern*. Um *graph pattern* é que define o critério de seleção das triplas, ou seja, define o cláusula *WHERE* de uma consulta SPARQL. Importante: em um *graph pattern* uma mesma variável pode figurar em vários *triple patterns* e, durante a execução da consulta, esta variável receberá o mesmo valor em todos os *triple patterns* em que aparece. A Tabela 5 apresenta a estrutura geral de uma consulta SPARQL SELECT.

Tabela 5 - Estrutura da consulta SPARQL SELECT

Estrutura de uma SPARQL SELECT *query*
diretiva base <URI> # lista de prefixos PREFIX pref: <URI> . . . # variáveis do resultado SELECT ... # grafo FROM ... # graph pattern de seleção WHERE { . . . } # modificadores ORDER BY ...

A Figura 24 mostra uma consulta SPARQL SELECT que retorna o título do livro.

Data:

```
<http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> "SPARQL Tutorial" .
```

Query:

```
SELECT ?title
WHERE
{
  <http://example.org/book/book1> <http://purl.org/dc/elements/1.1/title> ?title .
}
```

Query Result:

title
"SPARQL Tutorial"

Figura 24 - SPARQL SELECT (1) [Prud'hommeaux e Seaborne, 2008]

Um exemplo mais interessante é descrito na Figura 25. Devem ser retornados os valores da propriedades *foaf:name* e *foaf:mbox* de todos os recursos que possuam, ao mesmo tempo, estas duas propriedades. Neste caso, mais de um subgrafo satisfaz o critério de seleção (*graph pattern*) e, portanto, são retornados múltiplos resultados, um resultado para cada subgrafo.

Data:

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Johnny Lee Outlaw" .
_:a  foaf:mbox   <mailto:jlow@example.com> .
_:b  foaf:name   "Peter Goodguy" .
_:b  foaf:mbox   <mailto:peter@example.org> .
_:c  foaf:mbox   <mailto:carol@example.org> .
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

Query Result:

name	mbox
"Johnny Lee Outlaw"	<mailto:jlow@example.com>
"Peter Goodguy"	<mailto:peter@example.org>

Figura 25 - SPARQL SELECT (2) [Prud'hommeaux e Seaborne, 2008]

Uma nova versão da linguagem está sendo desenvolvimento (SPARQL 1.1[26]). Dentre as novas funcionalidades, destacam-se:

- **Funções de agregação**
  COUNT, SUM, MIN / MAX, AVG, etc.

- **Subqueries**
  Permitir usar o resultado de uma consulta para continuar outra consulta.

- **Negação**
  Facilitar a criação de expressões de negação, ou seja, aquelas que verificam que um dado *graph pattern* não está presente. Construções como NOT EXISTS, por exemplo.

[26] SPARQL 1.1 - http://www.w3.org/TR/sparql11-query/

- **Expressões na operação de projeção da SELECT *query***
  Permitir a criação de expressões para projetar qualquer expressão SPARQL, além de simples variáveis.

- ***Property Paths***
  Na criação de *triple patterns*, simplificar o acesso a propriedade desejada em caminhos hierárquicos.

- **Consulta Federada Básica**
  Consultas envolvendo múltiplas fontes de dados. Fontes da dados podem ser SPARQL *endpoints* remotos ou mesmo diferentes grafos (*named graphs*[27]) em um mesma fonte de dados.

Uma outra necessidade notável de SPARQL diz respeito às operações de atualização, ou seja, inserção e remoção de triplas. Até o momento para alterar a base dados RDF, temos que utilizar APIs de terceiros e fazê-lo programaticamente. No entanto, já existe uma proposta para atualizações via SPARQL chamada de SPARQL Update[28] (ou simplesmente, SPARUL) que também encontra-se na versão 1.1, e inclui, dentre outras, as seguintes facilidades: inserção e remoção de triplas, execução de um grupo de operações de atualização com uma única operação, e criação e remoção de grafos no banco de dados.

2.2.6. *Linked Data* e a Web de Dados

O conceito de *Linked Data* foi originalmente proposto por Tim Berners-Lee, em 2006 [Berners-Lee, 2006]. *Linked Data* refere-se a um conjunto de boas práticas para publicar e conectar dados estruturados na Web.

Na prática a ideia básica de *Linked Data* consiste em [Yu, 2011]:

- usar o modelo de dados RDF para publicar dados estruturados Web
- usar RDF links para interligar dados de diferentes fontes de dados

[27] Named Graphs - http://www.w3.org/2004/03/trix/

[28] SPARQL Update 1.1 - http://www.w3.org/TR/sparql11-update/

Aplicando este princípios nós iremos, paulatinamente, criar a chamada *Web de Dados* (também chamada de *Web of Linked Data*). A ideia é criar uma Web de dados estruturados interligados, de tal forma que máquina possa "navegar" por estes dados e compreendê-los. Seria para a máquina o análogo do que é a Web atual (agora chamada de Web de Documentos) para humanos.

Portanto pode-se dizer que *Linked Data* é um subconjunto da Web semântica. Como descrito anteriormente, o objetivo inicial da Web semântica era associar declarações semânticas aos documentos da Web. No entanto, logo percebeu-se que poderíamos publicar dados RDF independentemente dos Web sites tradicionais, criando uma nova Web, voltada para agentes de software processarem, bastando para isso interligar os dados RDF. Interligar dados RDF significa simplesmente unir grafos, ou seja, numa dada tripla o subjeito é uma URIref no *namespace* de um *dataset*, e o objeto é uma URIref no *namespace* de outro. Ao final teremos uma mega banco de dados RDF mundial.

Em [Berners-Lee, 2006], Tim Berners-Lee propõe quatro regras básicas para publicar e interligar dados RDF na Web. São elas:

1. Use URIs para nomear coisas.
2. Use HTTP URIs para que um cliente (homem ou máquina) possa acessar estes nomes na Web.
3. Toda URI deve ser dereferenciável. Em outras palavras, quando alguém acessar esta URI na Web, alguma informação útil deve sempre ser retornada.
4. Inclua links para outras URIs, para que o cliente pode descobrir novas coisas. Ou seja, navegar por RDF links entre diferentes fontes de dados.

Somando-se às quatro regras anteriores, vale ressaltar que é de extrema importância o reuso, sempre que possível, de URIs e de vocabulários pré-existentes, conhecidos pela comunidade, pois ajuda bastante no processo de interconexão dos grafos.

A Tabela 6 lista alguns destes vocabulários (ontologias) conhecidos e seus respectivos domínios. Uma lista mais completa destes vocabulários pode ser obtida no endereço:

http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/CommonVocabularies

Tabela 6 - Vocabulários comuns

Vocabulário	Domínio
Friend-of-a-Friend[29] (FOAF)	Descrever Pessoas.
Dublin Core[30] (DC)	Definir metadados genéricos.
Semantically-Interlinked Online Communities[31] (SIOC)	Representar comunidades online.
Description of a Project[32] (DOAP)	Descrever projetos.
Simple Knowledge Organization System[33] (SKOS)	Representar taxonomias e conhecimento fracamente estruturado.
Music Ontology[34]	Descrever artistas, álbuns e músicas.
Review Vocabulary[35]	Representar Revisões.
Creative Commons[36] (CC)	Descrever termos de licença.

Com respeito ao reuso de URIs, devemos considerar as URIs presentes nos *datasets* do projeto LOD (descrito a seguir), como Geonames[37], DBPedia[38], Musicbrainz[39], dbtune[40], RDF Mashup[41], entre outros.

O projeto *Linking Open Data Community* [LOD Project] tem contribuído significativamente para implementação da Web de Dados, interligando várias bases de dados RDF, de acordo com as regras anteriores. A Figura 26, conhecida como nuvem LOD (*LOD cloud diagram*), ilustra os vários datasets e suas interligações realizadas, até o momento, pelo projeto LOD. A cada dia esta nuvem cresce mais e mais.

[29] FOAF - http://xmlns.com/foaf/0.1

[30] DC - http://dublincore.org/documents/dcmes-xml

[31] SIOC - http://sioc-project.org

[32] DOAP - http://usefuline.com/doap

[33] SKOS - http://www.w3.org/2004/02/skos

[34] Music Ontology - http://musicontology.com

[35] Review Vocabulary - http://purl.org/stuff/rev#

[36] CC - http://creativecommons.org/ns

[37] Geonames - http://www.geonames.org/ontology

[38] DBPedia - http://dbpedia.org

[39] Musicbrainz - http://musicbrainz.org

[40] dbtune - http://dbtune.org

[41] RDF Mashup - http://www4.wiwiss.fu-berlin.de/bizer/bookmashup/index.html

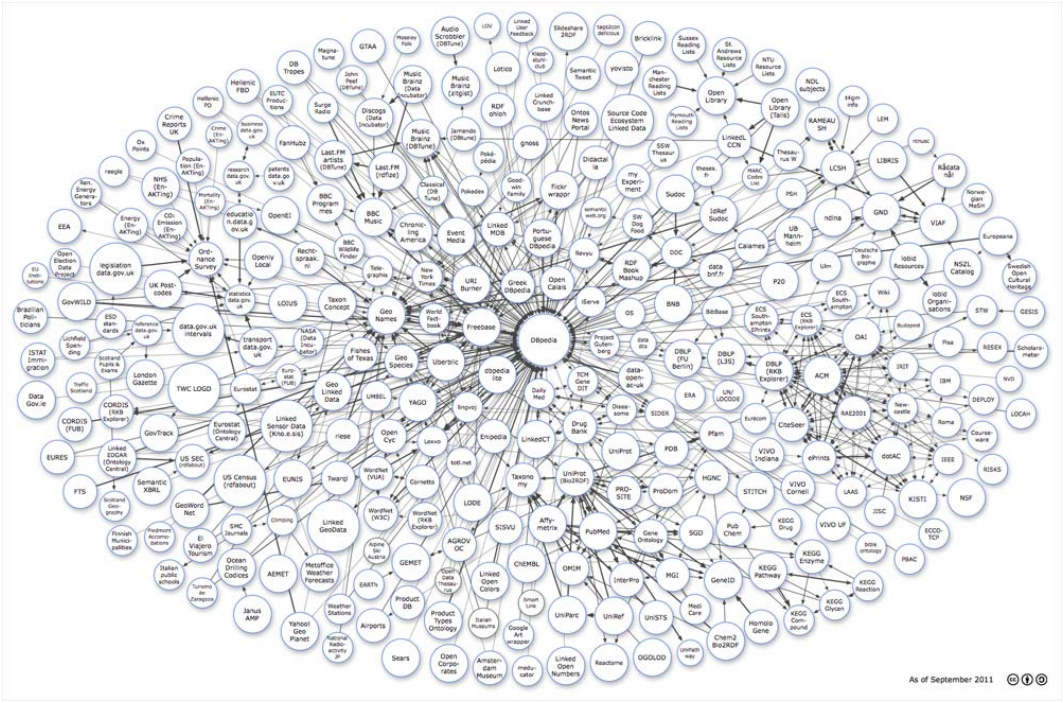

Figura 26 - LOD cloud diagram em Setembro de 2011 [LOD Project]

2.2.7. Gerenciamento de Transação em *Triple Stores*

Esta seção faz um levantamento do suporte às propriedades ACID oferecido pelos bancos de dados RDF mais comuns. Em especial com relação ao isolamento ou controle de concorrência, que é o foco deste trabalho. São eles:

- **Sesame2** (nativo)

  Sesame2 implementa atomicidade permitindo que a aplicação desabilite o *auto-commit* e chame *commit* ou *rollback* explicitamente.

  Ao inicializar o repositório nativo, Sesame2 cria um *Lock Manager* que gerencia bloqueios exclusivos e compartilhados. Se uma nova tripla for adicionada, uma bloqueio exclusivo é estabelecido em todo o repositório. Oferece o nível de isolamento *read committed*, permitindo leituras concorrentes, mas não escritas concorrentes.

- **Jena2**

  Implementa atomicidade, mas não é *thread-safe* por default. A aplicação tem que fazer seu próprio controle de concorrência. São

oferecidos bloqueios de escrita e leitura, usando como grânulo todo o repositório. É responsabilidade da aplicação respeitar o contrato, ou seja, ela não pode requisitar um bloqueio de leitura e, em seguida, realizar operações de escrita.

- **Bigdata**
  Implementa atomicidade e oferece controle de concorrência com multiversão, ou seja, nível de isolamento *snapshot*.

- **OWLIM**
  Implementa atomicidade e provê nível de isolamento *read committed*. No entanto, por questões de eficiência, e diferentemente do comportamento típico de banco de dados relacionais, modificações não confirmadas (*uncommitted*) não são visíveis mesmo usando a conexão que realizou as mudanças, ou seja, nem para a própria transação que realizou as mudanças. Portanto o seguinte trecho de código não funciona em OWLIM:

```
connection.add(subj, pred, obj);
assertTrue(connection.hasStatement(subj, pred, obj));
```

- **AllegroGraph**
  Implementa atomicidade e oferece controle de concorrência com multiversão, ou seja, nível de isolamento *snapshot*.

- **Virtuoso**
  Este faz jus ao nome, implementando todas as propriedades ACID e provendo os quatro níveis de isolamento (*read uncommitted, read committed, repeatable read e serializable)*. O nível de isolamento default *é read committed*.