Migrando do Firebird para o PostgreSQL. O que pode dar errado?

(c) Alexandr Shaposhnikov, editado e ajustado por Alexey Kovyazin

O material é protegido por lei de direitos autorais. A publicação e tradução são permitidas com a condição de ter um backlink claramente visível para a versão original e detalhes do autor original. Entre em contato com Alexey Kovyazin ak@firebirdsql.org com qualquer dúvida.

Table of Contents

Prefácio . . . 2
1. Diferenças na implementação MVCC . . . 3
  1.1. Teste 1 . . . 5
  1.2. Teste 2 . . . 6
2. SUSPEND e RETURN NEXT . . . 7
3. Usando Transações Autônomas . . . 8
4. Funcionalidade limitada do bloco PSQL anônimo no PostgreSQL . . . 9
5. Processos e Threads . . . 10
6. Contador de transações . . . 11
7. Trabalhando com sistemas de armazenamento . . . 12
8. Diferenças de backup . . . 13
  8.1. Gbak & pg_dump . . . 13
  8.2. Nbackup & pg_basebackup . . . 13
9. Problemas de migração de código envolvendo tabelas temporárias . . . 14
10. Comportamento do otimizador de consultas . . . 15
11. Tamanhos de dados . . . 16
12. Consequências de transações presas. VACUUM FULL e GFIX -sweep . . . 16
13. Uma transação por conexão . . . 17
14. Sem implementação de packages . . . 17
15. Modificando tabelas . . . 17
16. Verificação de dependências ao compilar procedimentos . . . 18
17. Migração de código SQL . . . 18
18. Incompatibilidade de tipos . . . 19
19. Diferentes implementações de triggers . . . 19
20. Ferramentas de banco de dados . . . 20
21. Administração mais complexa . . . 20
22. O hábito é uma segunda natureza . . . 20
23. Conclusão . . . 20
  23.1. Perguntas-chave a se fazer antes da migração . . . 21

24. Contatos 21

Prefácio

Não é segredo que, nos últimos anos, várias empresas decidiram com bastante frequência migrar um sistema de informação em funcionamento do Firebird para o PostgreSQL.

Uma situação típica se parece com isto:

O projeto está funcionando há vários anos. O cliente acredita que os problemas do projeto (todo software tem alguns problemas) são causados pelo SGBD. **Firebird é um SGBD "ruim".**

Em vez de:

- contratar empresas externas como consultores
- treinar e certificar seus próprios funcionários
- melhorar seu nível profissional

é muito mais fácil se convencer de que a causa raiz dos problemas é o Firebird, e decidir migrar para outro SGBD.

Este problema não está relacionado a um SGBD específico e é puramente gerencial.

Não acredita em mim? A linha destacada (Firebird é um SGBD "ruim".) é uma citação literal de uma das palestras dedicadas ao SGBD PostgreSQL, com uma mudança - no original, PostgreSQL foi escrito em vez de Firebird.

A situação é agravada porque os tomadores de decisão frequentemente não entendem os problemas e dificuldades deste processo.

Esta decisão causará sérios problemas para o sistema de TI da empresa. Normalmente, as pessoas seguem a regra simples "se algo funciona bem, deixe em paz." Esta regra teria impedido mudanças tão grandes. Mas, em vez disso, as pessoas acreditaram na ideia falsa de uma "bala de prata" - uma solução perfeita que pode magicamente consertar tudo.

Quem deveria ler este artigo

Este documento é para gerentes técnicos e CTOs que são responsáveis por projetos de migração de banco de dados. Projetos de migração criam sérios riscos para líderes técnicos porque eles frequentemente são culpados quando os custos ficam muito mais altos do que planejado (às vezes 2-3 vezes mais caros), quando os projetos demoram muito mais do que esperado, e quando os resultados finais não funcionam tão bem quanto prometido. Líderes técnicos devem saber que projetos de migração frequentemente falham, e gerentes de projeto geralmente são responsabilizados quando orçamentos crescem demais, cronogramas atrasam muito, e o novo sistema não performa tão bem quanto o antigo.

Lembre-se de que a dificuldade de migração depende de quão complexo é o seu banco de dados e quanto você usa recursos especiais do Firebird. O tempo e dinheiro necessários para migração mudam baseado em várias coisas: quanto código seu projeto tem e quão bom esse código é, quão

grande é o seu banco de dados, se você precisa tornar o sistema mais rápido, e outras questões que surgem durante o trabalho.

Este artigo não culpa o PostgreSQL! É sobre migração e suas consequências

PostgreSQL é um sistema de banco de dados maravilhoso com muitos recursos úteis, mas é diferente, e o comportamento do seu sistema mudará logo após a migração, e no início - ficará pior.

Quando você migra do Firebird para o PostgreSQL, você não obterá nenhum benefício dos recursos do PostgreSQL que o Firebird não tem durante a migração, mas definitivamente terá problemas com recursos ausentes do Firebird ou recursos que funcionam de forma diferente no PostgreSQL.

Consultas que funcionam bem no Firebird com os mesmos dados não funcionarão bem no PostgreSQL. O contrário também é verdade, mas ao migrar do Firebird para o PostgreSQL, você primeiro experimentará os problemas da primeira parte desta declaração.

Vamos dar uma olhada rápida nos problemas que você pode enfrentar durante esta migração.

1. Diferenças na implementação MVCC

Provavelmente a coisa mais desagradável e inesperada para aqueles que estão acostumados com o Firebird, ou melhor, com a forma como o Firebird implementa o versionamento de dados.

Embora seria mais correto falar sobre a implementação específica do MVCC no PostgreSQL: aqui o mecanismo de controle de versão é fundamentalmente diferente do Firebird, MS SQL, ORACLE.

Você pode ler sobre as diferenças neste artigo.

Em resumo, a abordagem clássica para implementar MVCC é implementar algum algoritmo "otimista", que assume que as transações serão completadas com sucesso, e versões antigas não serão usadas. Portanto, a nova versão do registro é escrita sobre a antiga, e ou o delta ou todo o registro antigo é escrito no log de desfazer, para que a versão antiga possa ser extraída quando necessário, o que deveria ocorrer raramente. Ou seja, a implementação clássica convencional do MVCC é armazenar diferenças e ressuscitar a versão antiga dos dados subtraindo diferenças da versão atual/última. É assim que funciona no Firebird.

Não é que a implementação MVCC do PostgreSQL seja ruim — é apenas fundamentalmente diferente. A ideia aqui é "copy-on-write", que cria versões de registro (cópias completas de registros com informações do sistema) muito rapidamente e em grandes números.

A implementação do MVCC no PostgreSQL não implica atualizar um registro; se uma atualização for necessária, exclusão e inserção são executadas.

Cada versão de um registro no PostgreSQL (tupla) é uma cópia completa do registro com alguns campos de serviço.

Por exemplo: o campo XMIN — ele contém o Id da transação que criou o registro, o que permite entender quais transações devem ver este registro. Ou o campo XMAX — ele contém o Id da transação que excluiu o registro.

O que acontece quando você atualiza um registro:

Primeiro, preenchemos o campo XMAX na versão atual do registro - onde escrevemos o Id da transação que atualiza, então criamos uma nova linha, na qual escrevemos o Id da transação que atualiza no XMIN. E um link para a nova é adicionado à versão antiga do registro.

Ou seja, na implementação - literalmente - DELETE E INSERT.

Quando você atualiza uma coluna de uma linha, toda a linha é copiada para a nova versão, provavelmente em uma nova página. (PostgreSQL tentará colocar a nova versão na mesma página onde a antiga estava, mas isso está longe de ser sempre possível), e a linha antiga também é alterada com um ponteiro para a nova versão. Registros de índice seguem o mesmo padrão: como há uma cópia completamente nova, todos os índices devem ser atualizados para apontar para a nova localização da página. **Todos os índices** - mesmo aqueles que não estão relacionados à coluna sendo alterada, são atualizados apenas porque toda a linha é movida.

Mais tarde, uma operação para limpar tuplas antigas (VACUUM) será necessária. Outra consequência desta abordagem é um grande volume de geração de WAL (log de refazer), porque muitos blocos são afetados quando uma tupla é movida para outra localização.

A consequência é uma carga de disco mais alta ao executar atualizações, maior intensidade durante a replicação.

Talvez a razão para a diferença na implementação do MVCC seja que o PostgreSQL começou e se desenvolveu como um projeto acadêmico, então a implementação do MVCC é honestamente acadêmica. Em outros sistemas, a implementação no momento da criação foi focada na eficiência, especialmente porque o desempenho dos computadores naquela época era significativamente menor do que agora.

Aqui você precisa mudar sua mentalidade. Depois de trabalhar com o Firebird, você espera que atualizar um registro custe muito menos que "excluir e inserir", mas lembre-se do velho ditado: "a realidade tem um jeito de esmagar expectativas", e o PostgreSQL segue suas próprias regras, não as suas.

Para ilustrar tudo o que foi dito acima, usando testes, criaremos uma tabela com a seguinte estrutura:

Tabela de Teste DAT (Sintaxe reduzida para legibilidade)

```
-- campos
CREATE TABLE DAT (
    ID BIGINT NOT NULL,
    I1..I8 BIGINT,
    N1..N8 DOUBLE PRECISION,
    D1..D8 TIMESTAMP,
    S1..S8 VARCHAR(100),
    T1..T8 VARCHAR(1000)
);

PK$DAT PRIMARY KEY (ID);

-- índices
```

```
X_I1 ON DAT (I1); X_I23 ON DAT (I2, I3); X_I456 ON DAT (I4, I5, I6);
X_N1 ON DAT (N1); X_N23 ON DAT (N2, N3); X_N456 ON DAT (N4, N5, N6);
X_D1 ON DAT (D1); X_D23 ON DAT (D2, D3); X_D456 ON DAT (D4, D5, D6);
X_S1 ON DAT (S1); X_S23 ON DAT (S2, S3); X_S456 ON DAT (S4, S5, S6);
X_F1 ON DAT (F1); X_F23 ON DAT (F2, F3); X_F456 ON DAT (F4, F5, F6);
X_T1 ON DAT (T1);
```

A tabela contém 10 milhões de registros, o primeiro campo do grupo está sempre preenchido, os campos restantes do grupo (No. 2-No. 8) são preenchidos com `NULL` com 50% de probabilidade.

A tabela de dados é idêntica nos bancos de dados Firebird 5.0.2 e PostgreSQL 17.4.

Os servidores de banco de dados estão rodando no Linux Mint 22.1 (SSD, 32Gb RAM).

Configuração básica dos servidores de banco de dados foi realizada.

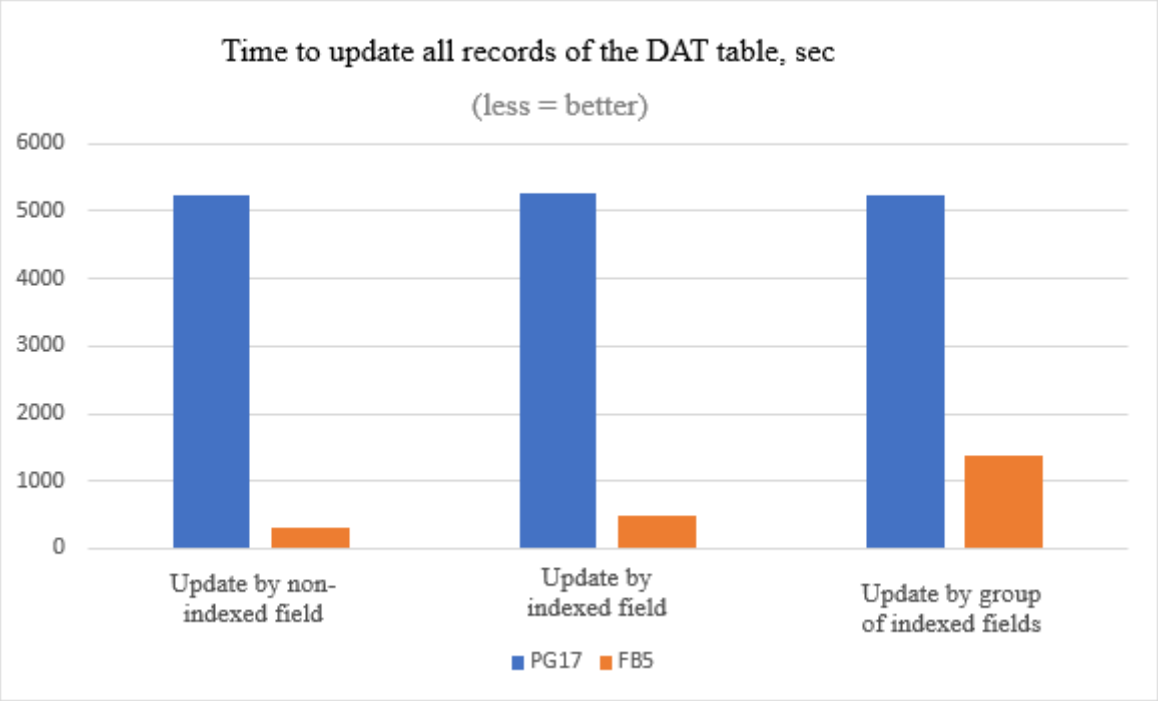
1.1. Teste 1

Ao atualizar um campo de tabela indexado no Firebird, apenas os índices que são construídos neste campo devem ser reconstruídos; ao atualizar no PostgreSQL, todos os índices devem ser reconstruídos, independentemente de a consulta `UPDATE` atualizar os campos nos quais esses índices são construídos ou não.

Vamos verificar isso:

10 milhões de registros atualizados	Firebird 5	PostgreSQL 17
por campo não-indexado `update dat set n8 = 0`	5 minutos 3 segundos	1 hora 27 minutos 23 segundos
por campo indexado `update dat set i4 = -i4`	8 minutos 16 segundos	1 hora 27 minutos 42 segundos

por grupo de campos indexados
`update dat`
`set i1 = 0, f2 = true, n4 = 4,`
`d6 = '1974-04-01', s3 = 'FC Bayern',`
`t1 = '1{...}Z'`
	23 minutos 16 segundos	1 hora 26 minutos 29 segundos



Vemos que os resultados práticos se correlacionam totalmente com a parte teórica: O tempo de atualização no PostgreSQL não depende de quantos campos atualizamos e se eles são indexados, no Firebird depende.

A diferença no tempo de execução de tais atualizações em massa no Firebird e PostgreSQL é muito significativa.

1.2. Teste 2

Vamos também comparar a velocidade de inserção e exclusão de registros:

Excluindo 1 milhão de registros de 10 milhões

```
delete from dat where id between 3000000 and 3999999
```

Firebird 5.0.2	PostgreSQL 17.4
2.4 segundos	2.2 segundos

Os dados médios para uma série de 50 medições são apresentados; a diferença é esperadamente insignificante.

INSERT 1 milhão de registros

```
insert into dat (id, i1, i2, i3, i4, i5, i6, i7, i8, f1, f2, f3, f4, f5, f6, f7, f8, n1, n2,
n3, n4, n5, n6, n7, n8, d1, d2, d3, d4, d5, d6, d7, d8, s1, s2, s3, s4, s5, s6, s7, s8, t1, t2,
t3, t4, t5, t6, t7, t8)
select id+10000000,  i1, i2, i3, i4, i5, i6, i7, i8, f1, f2, f3, f4, f5, f6, f7, f8, n1, n2,
n3, n4, n5, n6, n7, n8, d1, d2, d3, d4, d5, d6, d7, d8, s1, s2, s3, s4, s5, s6, s7, s8, t1, t2,
t3, t4, t5, t6, t7, t8 from dat where id between 1000000 and 1999999
```

Firebird 5.0.2	PostgreSQL 17.4
10 minutos 36 segundos	4 minutos 38 segundos

Aqui o resultado é notavelmente melhor para o PostgreSQL.

Conclusão: Você precisa analisar seu sistema de informação para identificar padrões de processamento de dados com funcionalidade similar antes da migração, decidir se eles são necessários, se podem ser substituídos de alguma forma ou completamente abandonados.

Adicionalmente, você precisa considerar que no PostgreSQL você começará a usar o dobro de espaço em disco por algum tempo após tal atualização e antes da coleta de lixo, que no Firebird também terá um efeito negativo muito menor — afinal, o espaço necessário para criar um delta ao atualizar um campo específico é significativamente menor do que criar uma cópia completa de um registro.

2. SUSPEND e RETURN NEXT

Trabalhando com Firebird, estamos acostumados com o fato de que ao executar um procedimento que retorna um número muito grande de linhas para uma aplicação cliente, o servidor Firebird dará a esta aplicação cliente alguma porção destes dados e parará até receber um comando da aplicação cliente sobre a necessidade de emitir uma nova porção. O suporte para tal comportamento é explicitamente implementado no código das partes cliente e servidor do Firebird. Em procedimentos armazenados, este comportamento é suportado usando o operador `SUSPEND`.

No PostgreSQL, este não é o caso

Da documentação:

> A implementação atual de `RETURN NEXT` e `RETURN QUERY` armazena todo o conjunto de resultados antes de retornar da função, como discutido acima. Isso significa que se uma função PL/pgSQL produzir um conjunto de resultados muito grande, o desempenho pode ser ruim: os dados serão escritos no disco para evitar exaustão de memória, mas a função em si não retornará até que todo o conjunto de resultados tenha sido gerado. Uma versão futura do PL/pgSQL pode permitir que os usuários definam funções que retornam conjuntos que não têm essa limitação. Atualmente, o ponto em que os dados começam a ser escritos no disco é controlado pela variável de configuração work_mem. Administradores que têm memória suficiente para armazenar conjuntos de resultados maiores na memória devem considerar aumentar este parâmetro.

Esteja preparado para o fato de que se o seu sistema tem procedimentos que atualmente não são totalmente buscados de acordo com a lógica de negócio da aplicação, então após a migração para PostgreSQL — eles começarão a ser. Isso pode levar à degradação do desempenho do sistema.

3. Usando Transações Autônomas

A principal (mas não a única) área de uso de transações autônomas é auditoria que não pode ser revertida.

Aqui está um exemplo de uso de uma transação autônoma para Firebird em um trigger em um evento de conexão de banco de dados para registrar todas as tentativas de conexão, incluindo as mal-sucedidas (Retirado da fonte Firebird 5.0 SQL Language Reference).

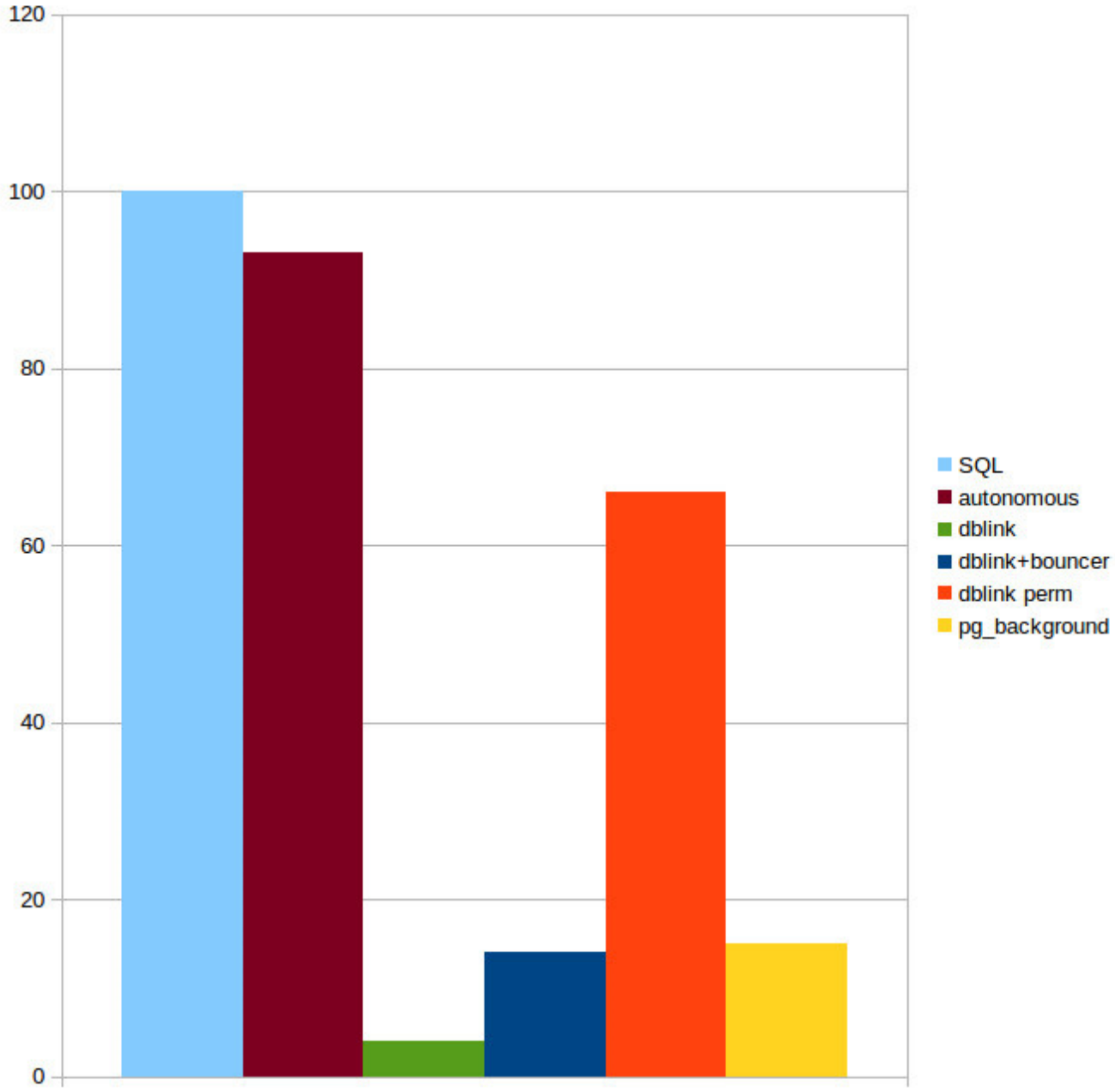
```
CREATE TRIGGER TR_CONNECT ON CONNECT AS
BEGIN
  -- Todas as tentativas de conectar ao banco de dados são salvas no log
  IN AUTONOMOUS TRANSACTION DO
    INSERT INTO LOG(MSG) VALUES ('USER ' || CURRENT_USER || ' CONNECTS.');
  IF (CURRENT_USER IN (SELECT USERNAME FROM BLOCKED_USERS)) THEN
  BEGIN
    -- salvar no log que a tentativa de conectar ao banco de dados foi malsucedida
    IN AUTONOMOUS TRANSACTION DO
    BEGIN
      INSERT INTO LOG(MSG) VALUES ('USER ' || CURRENT_USER || ' REFUSED.');
      POST_EVENT 'CONNECTION ATTEMPT' || ' BY BLOCKED USER!';
    END
    -- agora lançar uma exceção
    EXCEPTION EX_BADUSER;
  END
END
```

PostgreSQL vanilla não tem transações autônomas. Elas podem ser simuladas iniciando uma nova conexão usando `dblink` ou `pg_background`, mas isso resulta em sobrecarga, afeta o desempenho, e é simplesmente inconveniente, mas é factível. Aqui está um exemplo de implementação de alguma função de log usando a extensão `dblink`:

```
CREATE FUNCTION log_dblink(msg text) RETURNS void LANGUAGE sql
AS $function$
  select dblink('host=/var/run/postgresql port=5432 user=postgres dbname=postgres',
                format('insert into log select %L, %L', msg, clock_timestamp()::text))
$function$
```

Apenas o Postgres Pro (versão comercial do PostgreSQL), Enterprise Edition (nem mesmo Standard!) tem suporte completo para transações autônomas com execução de uma transação autônoma dentro do mesmo processo do servidor.

Apenas a última solução evita degradação de desempenho, porque é a única abordagem que executa uma transação autônoma dentro do mesmo processo do servidor.



O diagrama mostra o desempenho de uma consulta SQL regular (corresponde a 100% no diagrama), uma consulta em uma transação autônoma da versão Postgres Pro Enterprise e usando várias extensões (Dos materiais do Postgres Pro).

Se você usa ativamente transações autônomas no seu projeto Firebird para auditoria, leve isso em consideração.

4. Funcionalidade limitada do bloco PSQL anônimo no PostgreSQL

> Um bloco de código anônimo não recebe argumentos, e qualquer valor que possa retornar é descartado. Caso contrário, funciona como código de função.

> — Documentação PostgreSQL [(44.4. Anonymous Code Blocks)]

Claro, o problema com parâmetros de entrada pode ser resolvido de alguma forma usando algumas macros e substituindo-as por valores antes de executar a função, o conjunto resultante pode ser inserido em tabelas temporárias ou não logadas, mas primeiro, isso é inconveniente, e segundo, requer mudanças significativas no código e lógica da função.

`EXECUTE BLOCK` do Firebird não tem tais restrições, aqui está um exemplo desta construção sintática, calculando a média geométrica de dois números e retornando-a ao usuário:

```
EXECUTE BLOCK(x DOUBLE PRECISION=?, y DOUBLE PRECISION=?)
RETURNS (gmean DOUBLE PRECISION)
AS
BEGIN
  gmean = sqrt(x*y);
  SUSPEND;
END
```

Como você pode ver, o procedimento armazenado pode ser executado através desta construção com modificações mínimas no código fonte. Se a construção sintática `EXECUTE BLOCK` é usada com bastante frequência no seu sistema, então retrabalhar todo esse código durante a migração pode se tornar um problema.

5. Processos e Threads

PostgreSQL cria um novo processo (não thread) para cada conexão. Sem ajuste adequado e um servidor adequado, rajadas não planejadas de uso podem rapidamente sobrecarregar o banco de dados.

PostgreSQL tem usado o modelo de processo desde o início do projeto devido à sua simplicidade, e tem havido discussões contínuas sobre mudar para threads.

O modelo atual tem uma série de desvantagens: memória compartilhada estaticamente alocada não permite redimensionamento dinâmico de estruturas como o cache de buffer; algoritmos paralelos são difíceis de implementar e menos eficientes do que poderiam ser; sessões são fortemente vinculadas a processos. Usar threads parece promissor, embora seja repleto de dificuldades em isolamento, compatibilidade com sistemas operacionais, gerenciamento de recursos. Sem mencionar o fato de que a transição exigiria mudanças radicais no código e anos de trabalho. Até agora, a visão conservadora está ganhando, e nenhuma mudança é esperada no futuro próximo.

A implementação Linux de multitarefa é muito mais conducente a este comportamento, e é provavelmente a principal razão pela qual especialistas em PostgreSQL ~~recomendam,~~ ~~recomendam fortemente,~~ **forçam** o uso do PostgreSQL no Linux. Postgres Pro (distribuição comercial), por exemplo, abandonou completamente o suporte ao Windows nas versões recentes.

Firebird não é particularmente sensível a um grande número de conexões inativas, baseado na minha própria experiência de muitos anos operando um sistema de informação sob Firebird (e na arquitetura Classic Server, que é mais próxima à arquitetura PostgreSQL), posso dizer que nem 500,

nem 1000, nem 1500 conexões abertas causaram problemas de desempenho perceptíveis. A arquitetura Classic é precisamente "processo". As arquiteturas que fornecem trabalho com streams no Firebird são SuperClassic e SuperServer (esta arquitetura é recomendada nas versões modernas do Firebird).

Mudar o sistema operacional sob o qual seu SGBD opera significa substituir ou retreinar sua equipe, então se seu servidor Firebird está atualmente rodando Windows, é importante planejar isso com antecedência.

Se você é um fabricante de software distribuído, seus clientes terão que fazer uma transição similar, e é improvável que fiquem entusiasmados com isso.

Também note que independentemente do sistema operacional, PostgreSQL "não gosta" de um grande número de conexões simultaneamente abertas. Daí o pacote pgbouncer separado (fortemente recomendado para uso) para gerenciar o pool de conexões e o valor padrão do parâmetro `max_connections=100`.

No entanto, ao usar *pgbouncer* ou poolers de conexão similares, variáveis de contexto em nível de conexão não funcionarão adequadamente, o que pode exigir grandes mudanças em suas soluções que dependem deste recurso. Por exemplo, no Firebird você pode definir uma variável de conexão com o ID do usuário atual, que permite evitar passar o parâmetro de usuário para cada procedimento, mas no PostgreSQL você precisará usar uma abordagem diferente.

Também se tornará impossível usar tabelas temporárias em nível de conexão.

6. Contador de transações

O kernel do PostgreSQL tem um contador de transação de 32 bits, o que significa que não pode contar mais de 4 bilhões. Isso leva a problemas que são resolvidos por "congelamento" - um procedimento especial de manutenção de rotina chamado `VACUUM FREEZE`

Este é um "congelamento" agressivo de tuplas. Deve ser executado de tempos em tempos para todos os registros localizados longe no passado, e definir uma flag de que este registro está "congelado", ou seja, visível para todas as transações e tem um `FrozenTransactionId` especial (menos infinito).

O sistema entende que esta linha foi criada há muito tempo e o número da transação que a criou não importa mais. Isso significa que este número de transação pode ser reutilizado. Números de transação congelados podem ser reutilizados.

No entanto, se o contador transborda muito frequentemente, ou se transações congeladas impedem que o congelamento seja executado em tempo hábil, os custos deste procedimento são muito altos e podem até resultar na impossibilidade de escrever qualquer coisa no banco de dados.

No entanto, se você pode se dar ao luxo de parar seu cluster de BD a qualquer momento, iniciá-lo em modo de usuário único para executar o comando `VACUUM FREEZE` e esperar algumas horas, então este problema não te afeta.

Contadores de transação de 64 bits são um retrabalho fundamental do kernel do SGBD, e também são necessários apenas para sistemas altamente carregados, mas para eles não é apenas desejável.

É necessário.

É por isso que TODAS as versões comerciais do PostgreSQL agora implementam um contador de transação de 64 bits.

A implementação mais famosa e bem-sucedida é do Postgres Pro, que foram os primeiros no mundo a oferecer uma solução.

Há 4 razões pelas quais esta solução não está na versão vanilla:

1. Refatoração de uma grande quantidade de código — todo outro código PostgreSQL espera que o contador seja um número de 32 bits
2. Cada versão de linha tem um cabeçalho. Se o contador for de 64 bits, haverá muita informação de serviço para cada versão de linha.
3. Fazer tal mudança exigirá retrabalhar muitas extensões PostgreSQL, que devem entender que o número da transação agora é de 64 bits
4. "Inércia" da comunidade. A solução da Postgres Professional foi rejeitada pela comunidade devido ao seu tamanho e complexidade. O status atual é misterioso... "Retornado com feedback".

Quanto ao Firebird, quando o contador de transação está "esgotado", uma manutenção completa do BD será necessária, mas contadores de transação no Firebird desde a versão 3.0 (lançada em 2016) são de 48 bits e o problema descrito é mais teórico do que relacionado à vida real.

7. Trabalhando com sistemas de armazenamento

O mercado de sistemas de armazenamento de dados (aka SAN) mostra crescimento constante, onde os principais investimentos são observados do segmento corporativo. Programas de transformação digital e novos serviços online requerem atualizações constantes de infraestrutura, o que aumenta a demanda por sistemas de alto desempenho.

Se o seu sistema tem consultas que devem ser executadas muito rapidamente (dezenas de milissegundos), mas muito frequentemente, então você enfrentará um aumento perceptível no tempo total de execução de tais consultas após a migração se o banco de dados estiver localizado no SDS.

A razão é que o banco de dados PostgreSQL pode conter centenas de milhares de arquivos, e para cada arquivo, ao transferir informações entre o servidor e o SAN, custos adicionais surgirão para transferir dados sobre o nome do arquivo, informações de serviço, etc.

Muito grosseiramente, isso pode ser comparado à situação quando você copia um arquivo de 10 GB pela rede ou 1000 arquivos de 10 MB. Espero que todos conheçam a diferença de tempo entre essas operações.

Você pode ler mais sobre esta situação aqui.

8. Diferenças de backup

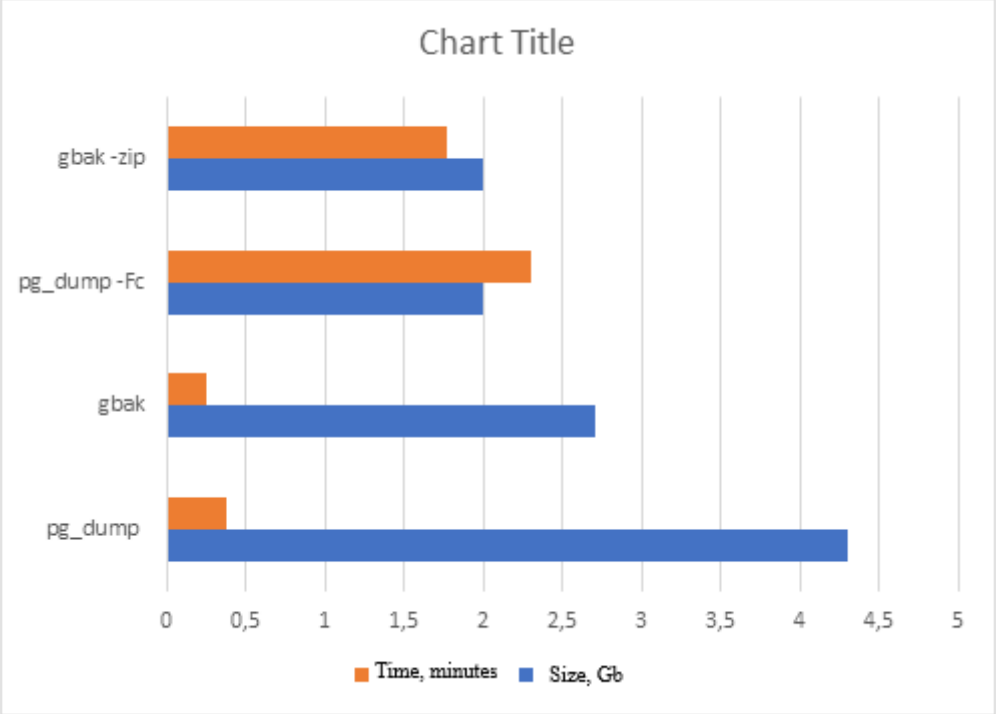
No PostgreSQL, backups funcionam completamente diferente de outros bancos de dados — tão diferente que requer uma mudança mental completa e abandonar tudo que você aprendeu sobre backups trabalhando com outros sistemas de banco de dados.

8.1. Gbak & pg_dump

Há um análogo do gbak para backup completo no PostgreSQL: pg_dump/pg_restore, e aqui tudo parece ser aproximadamente o mesmo:

- Backup de um banco de dados
- Você pode selecionar as tabelas necessárias
- Multi-threaded (no PostgreSQL — com restrições no formato do arquivo de saída)
- sem incremento
- Possibilidade de obter uma cópia comprimida

Obtendo uma cópia de backup usando o exemplo de um banco de dados com uma tabela `DAT`:



Mas `pg_dump` não é adequado para criar backups em tempo real de grandes bancos de dados, pois pode exigir recursos significativos e tempo de execução, o que pode afetar negativamente o desempenho do sistema. Com `gbak`, tal problema não foi observado desde o Firebird 2.5.

8.2. Nbackup & pg_basebackup

Para criar backups em tempo real, o Firebird tem `nbackup`, e o PostgreSQL tem `pg_basebackup`

E aqui, tudo é realmente diferente:

`pg_basebackup` sempre despeja todo o cluster, ao restaurar um banco de dados, todo o seu cluster PostgreSQL será restaurado, que conterá apenas um banco de dados. Os arquivos de todos os seus outros bancos de dados terão tamanho zero.

Desvantagens: capacidades limitadas ao trabalhar em modo multi-thread, limitado na criação de backups incrementais, menos flexibilidade comparado ao pg_dump na escolha de objetos individuais para backup, baixa velocidade de cópia.

`pg_probackup` da Postgres Professional tem muitas vantagens comparado ao `pg_basebackup`, mas a maioria delas está na versão comercial desta utilidade.

Na verdade, é bastante típico — quem pensaria em escrever uma versão comercial do nbackup do Firebird? Funciona perfeitamente!

Para fazer backups rápidos e confiáveis no PostgreSQL, você precisa entender o principal: o termo "banco de dados" no backup do PostgreSQL é um engano. Chame como quiser, mas trabalhe apenas com um cluster. Divida grandes bancos de dados em clusters separados em outras portas. Acredite em mim, é um mal menor (se você levar em conta o consumo de memória) do que manter todas as "bases" em um cluster.

Mesmo backup\restore frio não é tão simples, e backup\restore online via `pg_basebackup` é ainda mais difícil, já que é essencialmente levantar uma réplica.

`Nbackup` no Firebird é mais simples, mais compreensível e mais importante — mais conveniente.

9. Problemas de migração de código envolvendo tabelas temporárias

Há um problema no PostgreSQL que encontramos com bastante frequência. Infelizmente, a implementação da lógica de tabela temporária nele tem uma série de deficiências que afetam negativamente o desempenho do sistema.

Este fato é bem conhecido, assim como o fato de que a pergunta "Usamos tabelas temporárias no banco de dados SQL <ALGUM>, e não houve problemas. Por que isso acontece no PostgreSQL?" é bastante comum em fóruns.

- Criação ativa, modificação e exclusão de tabelas temporárias (incluindo trabalho com índices) pode levar a degradação significativa de desempenho
- AUTOVACUUM não vê tabelas temporárias — as operações correspondentes de limpeza e análise devem ser executadas manualmente
- Um padrão típico de trabalho com tabelas temporárias leva a um transbordamento do cache do SO com lixo
- pg_class (e outras tabelas do sistema) crescem durante a operação
- Proibição de escrever em tabelas temporárias em uma réplica ReadOnly (ao construir relatórios)

Considerando o uso generalizado de tabelas temporárias em soluções modernas, isso pode se tornar

um problema.

Soluções possíveis:

- Colocar tabelas temporárias em um disco RAM
- Usar tabelas não logadas (UNLOGGED) em vez de temporárias
- Usar versões comerciais do PostgreSQL, onde o mecanismo de tabelas temporárias foi significativamente redesenhado
- Limitar/evitar o uso de tabelas temporárias em suas aplicações

No Firebird:

- Tabelas temporárias são criadas fora do banco de dados em arquivos temporários
- Modo Forced Writes está sempre OFF para elas
- Elas são limpas instantaneamente

Não há problemas com o trabalho com tabelas temporárias, o que significa que durante a migração do sistema pode haver dificuldades causadas pelas especificidades acima mencionadas do PostgreSQL. Cuidado!

Notamos mais uma vez que se `pg_bouncer` for usado, então o código que anteriormente usava GTT em nível de conexão pode parar de funcionar.

10. Comportamento do otimizador de consultas

PostgreSQL tem um algoritmo de planejamento de consulta relativamente simples, mas rápido. No entanto, este algoritmo tem um grande problema. Ele constrói um plano, e então se apega a ele, mesmo se revelar-se errado. No pior caso, uma situação pode surgir quando o PostgreSQL espera 1-2 registros em cálculos intermediários, mas na verdade há milhares de vezes mais. Como resultado, executar um Nested Loop leva a enorme complexidade do algoritmo, que leva a um travamento do processo com alta carga de CPU. Infelizmente, por padrão, o PostgreSQL vanilla não tem a capacidade de especificar hints, como em alguns outros SGBDs.

Para o bem da objetividade, apontaremos a existência de extensões especiais para resolver este problema — por exemplo, pg_hint_plan.

O argumento é que "você precisa escrever consultas corretas, não corrigi-las". Bem, você pode ter que reescrever tais consultas problemáticas como "corretas" do ponto de vista do planejador PostgreSQL durante a migração.

Aqueles de vocês que migraram de uma versão do Firebird para outra sabem que o comportamento do planejador de consulta pode mudar — sem mencionar mudar planos de consulta ao mudar SGBD.

11. Tamanhos de dados

PostgreSQL não tem compressão em nível de página. Apenas dados TOAST são comprimidos. Se o banco de dados tem muitos registros com campos de texto relativamente pequenos, então a compressão poderia reduzir o tamanho do banco de dados várias vezes, o que não apenas economizaria em discos, mas também aumentaria o desempenho do SGBD. Consultas analíticas que leem muitos dados do disco e não os alteram com muita frequência podem ser aceleradas especialmente efetivamente reduzindo operações de entrada/saída.

Além disso, não podemos esquecer que em grandes bancos de dados de produção, o arquivo de transações do dia pode exceder o tamanho do banco de dados várias vezes.

A comunidade PostgreSQL sugere usar sistemas de arquivos com suporte a compressão para compressão. Mas isso nem sempre é conveniente e possível. Novamente, prepare-se para pagar vários milhares de USD por núcleo para versões comerciais que têm compressão de página!

Para testar esta tese, uma tabela de livros de referência combinados foi tirada de um banco de dados Firebird 5 de uma certa empresa de transporte e logística. Ela continha cerca de 700.000 linhas e muitos campos. Esta tabela foi convertida para um banco de dados PostgreSQL 17. A tabela continha os seguintes tipos de dados: strings, datas, flags, inteiros e números reais, uuids. A codificação de string é utf8.

O tamanho dos dados da tabela foi medido antes e depois da conversão.

Resultado: após a conversão, o tamanho ocupado pelos dados desta tabela no banco de dados aumentou exatamente uma vez e meia.

Note que antes do lançamento do Firebird 5, empacotar dados de texto com codificação utf8 no Firebird era notavelmente menos eficiente do que é agora.

Também deve ser lembrado que devido à implementação do MVCC, ao atualizar completamente um campo em toda a tabela, PostgreSQL exigirá espaço em disco igual ao tamanho de toda a tabela por algum tempo para executar a operação.

Dê uma olhada neste artigo com estudo e comparação mais detalhados: https://firebirdsql.org/img/articles/fb_vs_pg_datafill/fb_vs_pg_datafill.html

12. Consequências de transações presas. VACUUM FULL e GFIX -sweep

Alguns problemas no PostgreSQL que surgem devido a transações suspensas (inchaço de tabela e índice) são corrigidos apenas por `VACUUM FULL`, e ele:

- demora muito mais para executar do que `VACUUM`
- solicita um lock exclusivo de tabela
- requer espaço em disco (escreve uma nova cópia da tabela e não libera a antiga até que a operação seja completada)

O comando do Firebird `gfix -sweep` corrige problemas similares, típicos para arquitetura multi-versão, em modo online e é uma solução multi-thread e rápida.

Vale notar a extensão `pg_repack` do Postgres Pro comercial que executa reorganização de tabela sem um lock exclusivo.

13. Uma transação por conexão

Firebird suporta múltiplas transações por conexão. Se o seu sistema de informação rodando sob Firebird usa esta funcionalidade, ela deve ser reescrita — PostgreSQL não tem este recurso.

Tal funcionalidade permite executar múltiplas transações com diferentes níveis de isolamento dentro de uma única conexão.

A aplicação mais comum é ler dados em uma transação longa somente leitura, e alterar estes dados em transações curtas de escrita. Este comportamento sempre foi suportado por componentes de acesso Firebird populares, tanto desatualizados (FibPlus) quanto modernos (FireDac, UniDac). Não esqueça deste ponto ao migrar sua aplicação do Firebird para PostgreSQL.

14. Sem implementação de packages

Packages... Quão convenientes eles são durante o desenvolvimento. A capacidade de combinar procedimentos e funções por lógica de negócio ou tipo, a capacidade de atualizar todo o package, ao invés de atualizar um conjunto de procedimentos e funções... Por exemplo, temos um documento "Pedido ao Fornecedor" em nosso sistema. Implementamos todas as funções e procedimentos que trabalham com este documento dentro do package.

Atualizar o sistema após fazer mudanças é fácil — atualizamos este package inteiro no BD de destino, e não um conjunto de procedimentos e funções que mudamos ao lançar uma nova versão.

Você terá que abandonar packages após a migração — eles não estão disponíveis na versão vanilla do PostgreSQL, em versões comerciais eles são implementados através de esquemas e não são tão convenientes. Se você não usa packages no Firebird — é uma pena, porque é realmente muito conveniente. Se você os usa e vai migrar para PostgreSQL — também é uma pena, não apenas terá que reescrever tudo, mas também terá que esquecer packages.

15. Modificando tabelas

Se você quiser adicionar um campo a uma tabela após a migração, sem problemas! Mas apenas no final da tabela. Se você quiser que os campos na tabela estejam em uma certa ordem (ou seja, inserir um campo em algum lugar no meio da tabela), então isso só pode ser alcançado recriando e re-carregando completamente a tabela. Na verdade, no Firebird, em nível de formato, campos também são adicionados ao final da tabela. É só que as tabelas do sistema têm um campo adicional que é responsável pela ordem de saída do campo na IDE ou ao executar uma consulta `SELECT` com *. Parece ser uma pequena melhoria para o SGBD. Mas PostgreSQL não tem isso.

16. Verificação de dependências ao compilar procedimentos

Se você está acostumado a verificar dependências e metadados ao compilar procedimentos armazenados e triggers, abandone o hábito. No PostgreSQL, você descobrirá que cometeu um erro no nome da tabela no código do procedimento apenas quando ele for executado. Não muito agradável em produção.

Você simplesmente tem que aceitar esta realidade quando se move para PostgreSQL. Por exemplo, se consultas a tabelas específicas só rodam sob certas condições de negócio, quanto mais complexo o código do seu procedimento ou função se torna, maior a chance de conter um erro em nome de tabela ou view. Você só descobrirá este erro quando aquele caminho exato de código (e provavelmente muito raro) for executado.

17. Migração de código SQL

Mesmo se você não precisar reescrever muitos procedimentos armazenados, UDFs ou UDRs, você encontrará incompatibilidade de sintaxe SQL entre Firebird e PostgreSQL. Além disso, no estágio de migração, você não desfrutará do açúcar sintático que PostgreSQL tem mas Firebird não, mas encontrará dificuldades porque PostgreSQL carece de algumas facilidades do Firebird ou a presença de algumas construções sintáticas específicas do Firebird em consultas (por exemplo, `SELECT FIRST … SKIP`).

O que mais sinto falta é a construção sintática muito conveniente do Firebird `UPDATE OR INSERT`, cuja implementação no PostgreSQL via `on conflict…` é, na minha opinião, longe de conveniente.

Implementação de UPSERT no PostgreSQL:

```
INSERT INTO director
  (id, name, fd1, fd2, fd3, fd4, fd5)
  VALUES (1, 'director Name',1,2,3,4,5)
ON CONFLICT (id)
DO UPDATE
SET name = EXCLUDED.name
  , fd1 = EXCLUDED.fd1
  , fd2 = EXCLUDED.fd2
  , fd3 = EXCLUDED.fd3
  , fd4 = EXCLUDED.fd4
  , fd5 = EXCLUDED.fd5;
```

Compare com a implementação Firebird:

```
update or insert
into director (id, name, fd1, fd2, fd3, fd4, fd5)
VALUES (1, 'director Name',1,2,3,4,5)
matching (Id)
```

Ao adicionar a cláusula `returning old.id, new.id into :old_id, :new_id` à construção especificada no procedimento, seremos capazes de determinar se a próxima linha foi inserida ou atualizada.

18. Incompatibilidade de tipos

Postgres tem significativamente mais tipos de dados, mas isso é de pouca ajuda durante a migração. Mas a presença de tipos Firebird, que no Postgres precisarão ser declarados de forma diferente (BLOB, DECIMAL, INT128, DECFLOAT ...) pode causar algumas dificuldades.

Particularmente digno de nota é o tipo DECFLOAT, que é um tipo numérico do padrão SQL:2016 e armazena com precisão números de ponto flutuante. Diferente do DECFLOAT, os tipos FLOAT ou DOUBLE PRECISION fornecem uma aproximação binária da precisão esperada. Pelo que eu sei, apenas Firebird e DB2 têm uma implementação deste tipo.

19. Diferentes implementações de triggers

Um trigger no PostgreSQL consiste de dois componentes: uma condição que determina quando o trigger deve disparar, e uma ação (função de trigger) que deve ser executada quando o trigger dispara. No Firebird, este é um componente-função único. A implementação de triggers no PostgreSQL é mais ampla e funcional, mas durante a migração, é uma reescrita de código. Se o seu sistema tem um grande número de triggers, então sua reescrita levando em conta as especificidades do PostgreSQL pode levar algum tempo.

```
CREATE TABLE orders (
  order_id bigint, name VARCHAR(100),
  quantity INT, updated_at TIMESTAMP
);
```

Trigger no Firebird:

```
CREATE OR ALTER TRIGGER ORDERS_TBI FOR ORDERS
ACTIVE BEFORE INSERT POSITION 0 AS
begin
  new.updated_at = localTimestamp;
end
```

Trigger no PostgreSQL:

```
CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $
BEGIN
    NEW.updated_at := NOW();
    RETURN NEW;
END;
$ LANGUAGE plpgsql;

CREATE TRIGGER before_update_orders
```

```
BEFORE UPDATE ON orders
FOR EACH ROW
  EXECUTE FUNCTION set_updated_at();
```

20. Ferramentas de banco de dados

Se você precisar de um gerenciador para trabalhar com PostgreSQL, focado em desenvolver e debugar procedimentos armazenados, então sua escolha: "EMS SQL Manager for Postgres SQL", "Postgres SQL Maestro" ou "Navicat Premium". Todas elas são muito inferiores às ferramentas que estamos acostumados ao trabalhar com Firebird.

21. Administração mais complexa

Esteja preparado para o fato de que você precisará de recursos humanos adicionais para a administração e suporte de seus bancos de dados após a migração. PostgreSQL é mais difícil de administrar que Firebird e muito rapidamente se torna coberto com vários utilitários e complementos — `pgbouncer`, `pgpool`, `patroni` e dezenas de outros que também precisam de "supervisão".

Como resultado, o orçamento mensal para desenvolvimento e manutenção do sistema pode aumentar significativamente.

22. O hábito é uma segunda natureza

Diferente do Firebird, no código de procedimentos e funções PostgreSQL, parâmetros de entrada-saída não são marcados com ":". Quando ambiguidade ocorre, isso leva ao erro "column is ambiguous" e no início causa um leve estupor e desconforto - como lidar com isso?

Como superar este desconforto pode ser encontrado neste artigo.

Acostumar-se com o fato de que todo IF requer um END IF também levará tempo. No início, você constantemente esquece disso, e o fato de que este erro é detectado em qualquer lugar, mas não onde você esqueceu de colocar este `END IF`, não ajuda muito.

23. Conclusão

Aviso Importante: Este artigo não pretende criticar o PostgreSQL. PostgreSQL é um excelente sistema de gerenciamento de banco de dados com muitos recursos avançados que o Firebird não tem (assim como o Firebird é muito bom e tem muitos recursos que o PostgreSQL não tem).

Quando seu sistema de informação foi projetado e desenvolvido no ambiente Firebird, ele se torna parte daquele ecossistema. Pense nisso como uma planta que cresce em um clima específico. Se você de repente mover essa planta para um clima completamente diferente, ela enfrentará sérios desafios para sobreviver.

A mesma coisa acontece com migração de banco de dados. Seu sistema se adaptou para trabalhar

com recursos específicos, comportamentos e características do Firebird. Uma mudança súbita para PostgreSQL (ou qualquer outro SGBD) definitivamente causará problemas e complicações.

23.1. Perguntas-chave a se fazer antes da migração

Antes de iniciar qualquer projeto de migração, você deve ter razões claras e fortes. Faça a si mesmo estas perguntas importantes:

1. Qual é o Problema Real?

1. Você está enfrentando limitações técnicas reais com Firebird? Crie uma lista das limitações e verifique-a contra a documentação. Descubra se outras empresas ou desenvolvedores enfrentaram as mesmas limitações e como as resolveram.
2. Seu banco de dados atual está falhando em atender requisitos de negócio? Determine o que exatamente está falhando.
3. Você está experimentando problemas de desempenho que não podem ser resolvidos no Firebird? Certifique-se de ter estreitado os problemas ao nível do Firebird, exclua configurações de hardware, VM e SO.

2. Quais são seus benefícios esperados?

1. A migração resolverá seus problemas atuais?
2. Que novas capacidades você ganhará?
3. Como isso melhorará suas operações de negócio?
4. Você pode medir o retorno esperado do investimento na migração?

3. Você considerou todas as alternativas?

1. Você pode atualizar sua versão atual do Firebird?
2. Existem soluções baseadas em Firebird para seus problemas?
3. Você otimizou seu design atual de banco de dados?
4. Ferramentas de terceiros poderiam ajudar sem migração?
5. Você pediu ajuda profissional?

24. Contatos

Sinta-se à vontade para contatar Alexey Kovyazin com todas as perguntas: ak@firebirdsql.org.